Grundlagen der Theoretischen Informatik

Folien erstellt von Dietmar Saupe und Marcel René Hiller
Version vom SS 2014, 29.07.2015

Universität Konstanz, Germany

Auf der Grundlage von
Uwe Schöning, Theoretische Informatik — kurzgefasst, Spektrum Akademischer Verlag, 5. Auflage, 2008.

Vorbemerkungen

- Diese Folien dienen zur Ergänzung der Vorlesung “Theoretische Grundlagen der Informatik”, Universität Konstanz, SS 2014.
- Als Vorlesungsskript dient das Buch “Theoretische Informatik — kurzgefasst” von Uwe Schöning, Spektrum Akademischer Verlag, 5. Auflage, 2008 (ca. 200 Seiten).
- Als Ergänzung empfohlen: “Einführung in Automatentheorie, formale Sprachen und Berechenbarkeit”, Hopcroft, Matwani, Ullman, 3. Auflage, Pearson, 2011 (ca. 600 Seiten, Musterlösungen online).
- Diese Foliensammlung deckt weder die gesamte Vorlesung noch das gesamte Skript ab.

Übersicht Themen

Automatentheorie Studium abstrakter Rechengeräte (“Maschinen”)

Turing-Maschine Mathematisches Modell einer universellen Maschine (A. Turing, 1930’er). Was ist prinzipiell berechenbar?

Endliche Automaten Spezielle, einfache, nützliche Automaten

Grammatiken Beschreibung von “Sprachen”, eng verwandt mit Automaten, Grundlage von (Teilen von) Compilern (N. Chomsky, 1950’er)

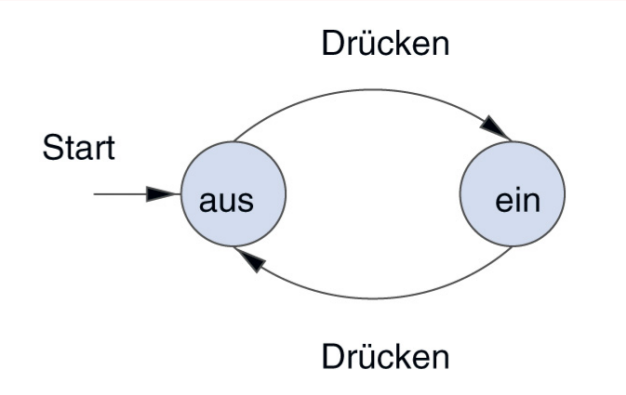
Komplexität Was ist effizient berechenbar, bzw. was ist *nicht* handhabbar (NP-hart, intractable)?

$\rightarrow$ Grundlagen von Software, der Entwicklung von Algorithmen

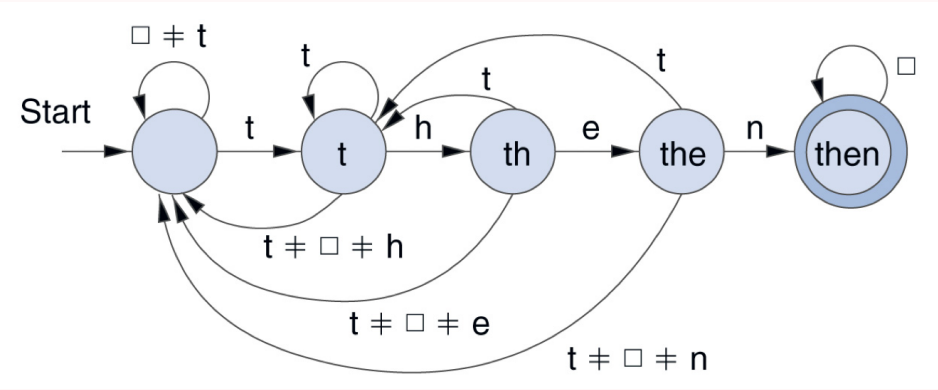
Einsatz endlicher Automaten

- Entwurf/Prüfung digitaler Schaltkreise
- lexikalische Analysekomponente von Compilern (logische Einheiten, z.B. Bezeichner, Schlüsselwörter, Satzzeichen)
- Suche in großen Textkorpora (z.B. Webseitensammlungen) nach Wörtern, Ausdrücken und Mustern
- Verifikation von Systemen mit endlich vielen Zuständen (z.B. Protokolle zu Kommunikation)

Kippschalter



Erkennung Schlüsselwort *then*



Bilder aus Hopcroft, Motwani, Ullman, s.o.

Grammatiken Modelle für Software zur Verarbeitung von Daten mit rekursiver Struktur. Beispiel: Parser. Regelbeispiel $E \to E + E$. ($E =$ Expression)

Reguläre Ausdrücke Struktur von (Text-)Zeichenreihen. Beispiel:

"[A-Z][a-z]$*$[ ][A-Z][A-Z]"

repräsentiert US-Städtenamen inkl. Bundesstaatsangabe wie z.B. "Orlando FL"

"[ ]": 1 Zeichen, also [A-Z] = 1 (großer) Buchstabe
"$*$": eine beliebige Anzahl

Frage: US Städte mit mehreren Wörtern (z.B. New York City NY)?
Antwort: "[A-Z][a-z]$*$([ ][A-Z][a-z]$*$)$*$[ ][A-Z][A-Z]"

Achtung: Auch "London UK" und "A B Cd EF" gehört dazu.

Zwei Fragen zu den Grenzen der Berechenbarkeit:

1. Was können Computer *überhaupt* leisten?
   Frage der “Entscheidbarkeit”.
2. Was können Computer *effizient* leisten?

   Betrachte die Laufzeit als Funktion $f(n)$ der Eingabelänge n.
   Problem ist handhabbar (effizient lösbar) wenn es ein (deterministisches) Verfahren mit polynomialer Laufzeit $f(n)$ gibt.
   Wenn $f(n)$ stets größer (d.h. exponentiell) ist, dann ist das Problem nicht handhabbar (NP-hart).

   Informatik liefert Methode mit der der Nachweis dafür erbracht wird, d.h. es werden Aussagen zu allen Algorithmen zu einem Problem bewiesen, inklusive aller noch nicht erfundenen Algorithmen!

Übersicht

Formale Sprachen

- Sei Σ ein Alphabet.
- Eine (formale) *Sprache* (über Σ) ist eine beliebige Teilmenge von Σ^*.

Beispiel

- $\Sigma = \{(,), +, -, *, /, a\}$.
- Sprache der korrekt geklammerten arithmetischen Ausdrücke $EXPR \subseteq \Sigma^*$:

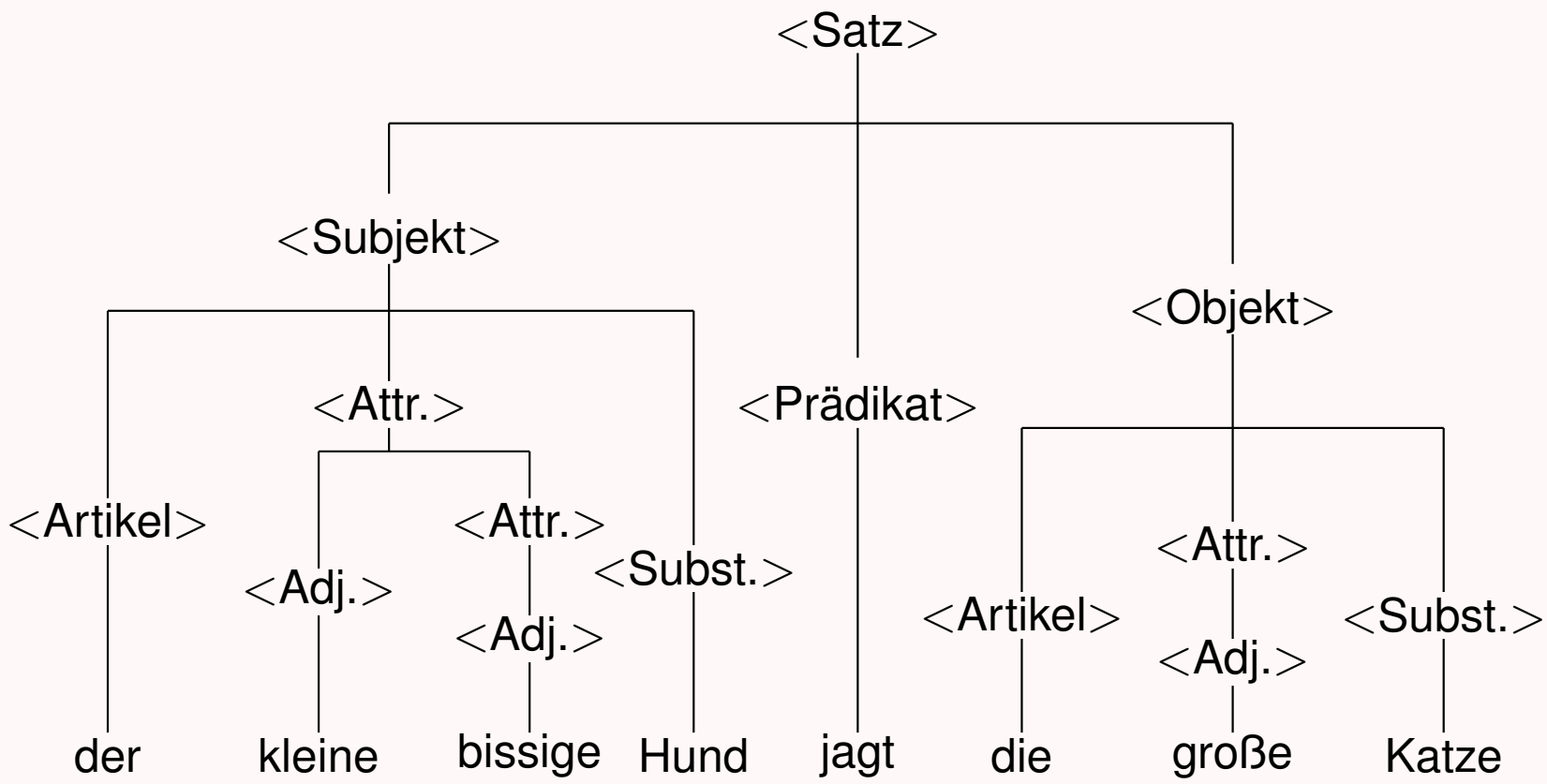
$$\begin{aligned} (a-a)*a+a/(a+a)-a &\in EXPR \\ (((a))) &\in EXPR \\ ((a+)-a(&\notin EXPR \end{aligned}$$

- a ist Platzhalter für beliebige Konstanten oder Variablen.

Endliche Beschreibung einer Sprache: Beispiel einer Grammatik

<Satz>	$\rightarrow$	<Subjekt> <Prädikat> <Objekt>
<Subjekt>	$\rightarrow$	<Artikel> <Attribut> <Substantiv>
<Artikel>	$\rightarrow$	ε
<Artikel>	$\rightarrow$	der
<Artikel>	$\rightarrow$	die
<Artikel>	$\rightarrow$	das
<Attribut>	$\rightarrow$	ε
<Attribut>	$\rightarrow$	<Adjektiv>
<Attribut>	$\rightarrow$	<Adjektiv> <Attribut>
<Adjektiv>	$\rightarrow$	kleine
<Adjektiv>	$\rightarrow$	bissige
<Adjektiv>	$\rightarrow$	große
<Substantiv>	$\rightarrow$	Hund
<Substantiv>	$\rightarrow$	Katze
<Prädikat>	$\rightarrow$	jagt
<Objekt>	$\rightarrow$	<Artikel> <Attribut><Substantiv>

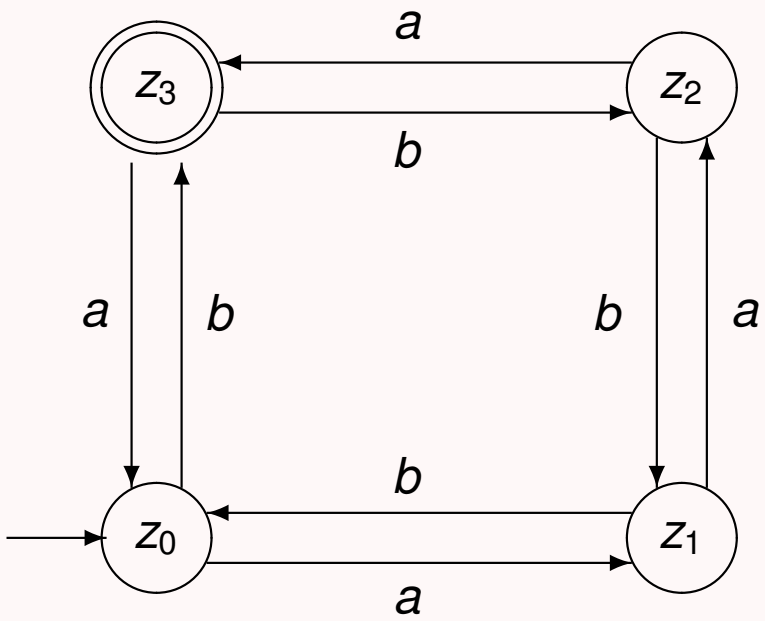
Ableitung eines “Satzes” als *Syntaxbaum*



Endliche Beschreibung einer Sprache: Beispiel eines Automaten

- Zustände z_0, z_1, z_2, z_3
- Alphabet $\Sigma = \{a, b\}$, Sprache $L \subseteq \Sigma^\star$
- $aaa, abaaa, b, bba, abbababbaabbbaa \in L$

Zustandsgraph eines Automaten



Definition Grammatiken

$\rightarrow$ Tafelaufschrieb

Beispiel 1: korrekt geklammerte arithmetische Ausdrücke

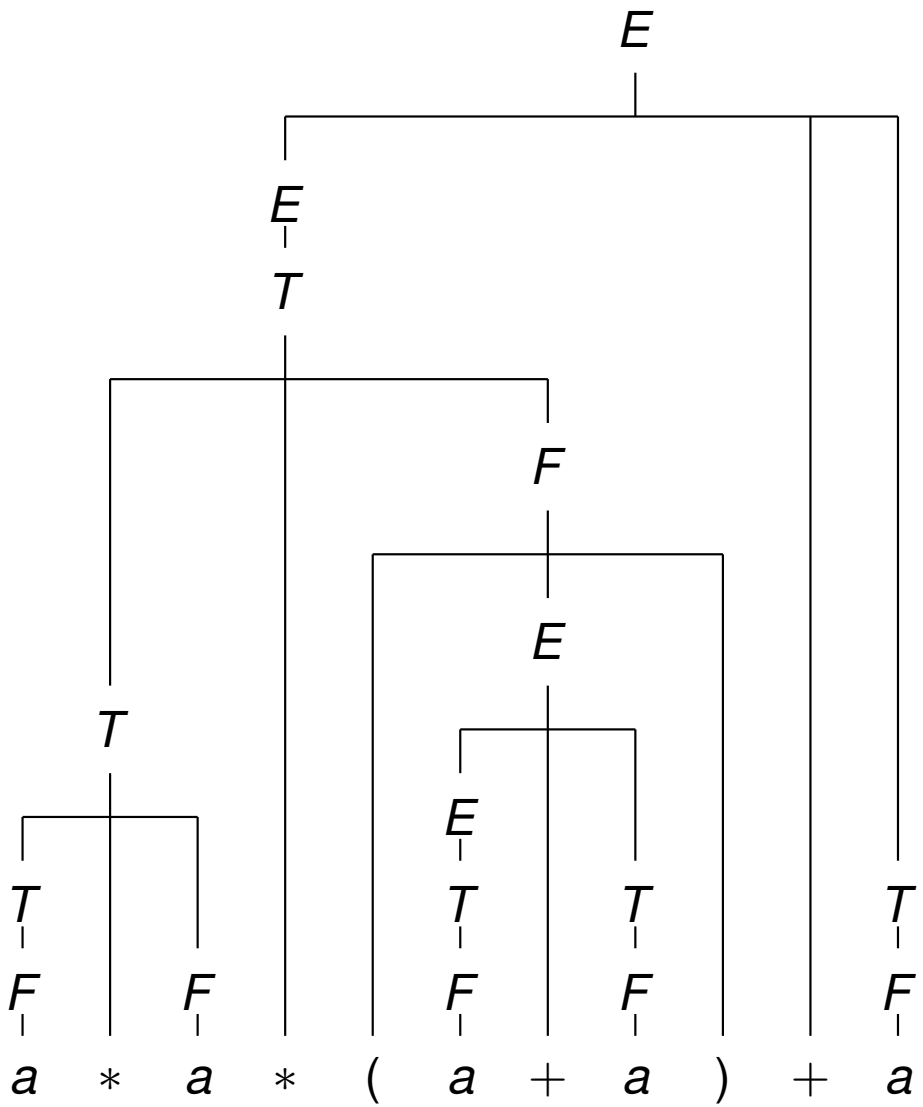
$G = (\{E, T, F\},\ \{(,), a, +, *\},\ P,\ E)$ wobei

$$\begin{aligned} P \quad = \quad \{\ & E \to T, \\ & E \to E + T, \\ & T \to F, \\ & T \to T * F, \\ & F \to a, \\ & F \to (E)\ \} \end{aligned}$$

Linksableitung von $a * a * (a + a) + a \ \in \ L(G)$ ist:

$$\begin{aligned} & E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * F + T \Rightarrow T * F * F + T \Rightarrow \\ & F * F * F + T \Rightarrow a * F * F + T \Rightarrow a * a * F + T \Rightarrow \\ & a * a * (E) + T \Rightarrow a * a * (E + T) + T \Rightarrow a * a * (T + T) + T \Rightarrow \\ & a * a * (F + T) + T \Rightarrow a * a * (a + T) + T \Rightarrow a * a * (a + F) + T \Rightarrow \\ & a * a * (a + a) + T \Rightarrow a * a * (a + a) + F \Rightarrow a * a * (a + a) + a \end{aligned}$$

Syntaxbaum zum Beispiel 1



Beispiel 2: $L(G) = \{a^n b^n c^n \mid n \geq 1\}$

Grammatik

$$
\begin{aligned}
G &= (V, \Sigma, P, S)\ , \text{ wobei:} \\
V &= \{S, B, C\} \\
\Sigma &= \{a, b, c\} \\
P &= \{S \to aSBC, S \to aBC, CB \to BC, \\
&\qquad aB \to ab, bB \to bb, bC \to bc, cC \to cc\}
\end{aligned}
$$

Ableitung von $a^3 b^3 c^3$

$$
\begin{aligned}
&\underline{S} \Rightarrow \overline{aSBC} \Rightarrow a\overline{aSBC}BC \Rightarrow aa\overline{aBC}\underline{B}CBC \Rightarrow \\
&aaaB\overline{BC}\underline{CB}C \Rightarrow aaaBB\underline{C}\overline{BC}C \Rightarrow aa\underline{a}BB\overline{BC}CC \Rightarrow \\
&aa\overline{ab}\underline{B}BCCC \Rightarrow aaab\overline{b}\underline{B}CCC \Rightarrow aaabb\overline{b}\underline{C}CC \Rightarrow \\
&aaabb\overline{bc}\underline{C}C \Rightarrow aaabbb\overline{cc}\underline{C} \Rightarrow aaabbbcc\overline{c} = a^3 b^3 c^3
\end{aligned}
$$

Beweis von $L(G) = \{a^n b^n c^n \mid n \geq 1\}$

$L(G) \supseteq \{a^n b^n c^n \mid n \geq 1\}$

- Beispiel für $n = 3$ offensichtlich für alle $n \geq 1$ anwendbar

$L(G) \subseteq \{a^n b^n c^n \mid n \geq 1\}$

- Regel erhalten Balance: $\#\{a, A\} = \#\{b, B\} = \#\{c, C\}$
- 'a' kann nur ganz links oder rechts von 'a' erzeugt werden
- 'b' kann nur rechts von 'a' oder 'b' erzeugt werden
- 'c' kann nur rechts von 'b' oder 'c' erzeugt werden

Ableiten ist nicht deterministisch

- Die Ableitungsrelation $\Rightarrow_G$ ist i.A. keine (partielle) Funktion:
- Zu $x \in \{V \cup \Sigma\}^\star$ kann es mehrere x' geben mit $x \Rightarrow_G x'$
  - Verschiedene Produktionen anwendbar auf ein Teilwort
  - Verschiedene Teilwörter sind linke Seite von Produktionen

Algorithmus für Wortproblem für kontextsensitive Sprachen

INPUT (G, x); $\{\ |x| = N\ \}$
$T := \{S\}$;
REPEAT
  $T_1 := T$;
  $T := Abl_N(T_1)$
UNTIL $(x \in T)$ OR $(T = T_1)$;
IF $x \in T$
  THEN WriteString('x liegt in $L(G)$')
  ELSE WriteString('x liegt nicht in $L(G)$')
END

Bemerkungen

- Laufzeit: exponentiell in N
- Wortproblem für kontextsensitive Sprachen ist *NP-hart*

Beispiel Wortproblem

Bekannte Grammatik für $L(G) = \{a^n b^n c^n \mid n \geq 1\}$

$$S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$$

Ausführung des Algorithmus' für Wortlänge 4

- $$\begin{aligned} T_0^4 &= \{S\} \\ T_1^4 &= \{S, aSBC, aBC\} \\ T_2^4 &= \{S, aSBC, aBC, abC\} \\ T_3^4 &= \{S, aSBC, aBC, abC, abc\} \\ T_4^4 &= \{S, aSBC, aBC, abC, abc\} = T_3^4 \end{aligned}$$
- Es gibt nur ein Wort der Länge ≤ 4 in L: $abc \in L$

(Erweiterte) Backus-Naur Form für kontextfreie Grammatiken

$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$$\begin{aligned} A &\to \beta_1 \\ A &\to \beta_2 \\ &\vdots \\ A &\to \beta_n \end{aligned}$$

$A \to \alpha[\beta]\gamma$

$$\begin{aligned} A &\to \alpha\gamma \\ A &\to \alpha\beta\gamma \end{aligned}$$

$A \to \alpha\{\beta\}\gamma$

$$\begin{aligned} A &\to \alpha\gamma \\ A &\to \alpha B\gamma \\ B &\to \beta \\ B &\to \beta B \end{aligned}$$

Bemerkungen

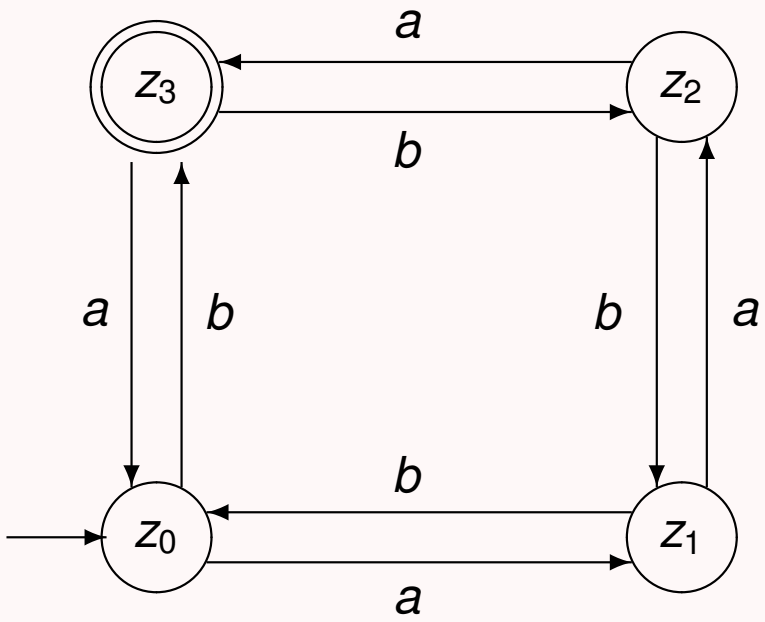
- *(E)BNF* Form
- Grammatiken in (E)BNF Form sind genau die Typ 2 Grammatiken (kontextfreie Sprachen).

Beispiel eines deterministischen endlichen Automaten

$M = (Z, \Sigma, \delta, z_0, E)$

$$
\begin{aligned}
Z &= \{z_0, z_1, z_2, z_3\} \\
\Sigma &= \{a, b\} \\
E &= \{z_3\} \\
\delta(z_0, a) &= z_1 \\
\delta(z_0, b) &= z_3 \\
\delta(z_1, a) &= z_2 \\
\delta(z_1, b) &= z_0 \\
\delta(z_2, a) &= z_3 \\
\delta(z_2, b) &= z_1 \\
\delta(z_3, a) &= z_0 \\
\delta(z_3, b) &= z_2
\end{aligned}
$$

Zustandsgraph



Beschreibungsmittel für Typen von Sprachen

Typ 3	reguläre Grammatik DFA NFA regulärer Ausdruck
Det. kf.	$LR(k)$-Grammatik deterministischer Kellerautomat (DPDA)
Typ 2	kontextfreie Grammatik Kellerautomat (PDA)
Typ 1	kontextsensitive Grammatik linear beschränkter Automat (LBA)
Typ 0	Typ 0 - Grammatik Turingmaschine (TM)

Determinismus und Nichtdeterminismus

Nichtdet. Automat	Determ. Automat	äquivalent?
NFA	DFA	ja
PDA	DPDA	nein
LBA	DLBA	?
TM	DTM	ja

Die Frage, ob sich nichtdeterministische LBAs äquivalent in deterministische umformen lassen, ist ungelöst. Diese Frage ist als *LBA-Problem* bekannt.

Abschlusseigenschaften

	Schnitt	Vereinigung	Komplement	Produkt	Stern
Typ 3	ja	ja	ja	ja	ja
Det. kf.	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 0	ja	ja	nein	ja	ja

Entscheidbarkeit

	Wort-problem	Leerheits-problem	Äquivalenz-problem	Schnitt-problem
Typ 3	ja	ja	ja	ja
Det. kf.	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

Wortproblem

Typ 3 (DFA gegeben)	lineare Komplexität
Det. kf.	lineare Komplexität
Typ 2 (CNF gegeben)	Komplexität: $O(n^3)$
Typ 1	exponentielle Komplexität NP-hart
Typ 0	unlösbar

Intuitive Berechenbarkeit

- Berechenbarkeit von Funktionen soll *mathematisch* definiert werden
- Zweck: Beweisbarkeit von Berechenbarkeit bzw. Nicht-Berechenbarkeit

Intuitive Definition

Eine Funktion $f : N^k \to N$ (bzw. eine partielle Funktion $f : N^k \to N \cup \{\text{undef}\}$) soll als *berechenbar* gelten, falls es einen Algorithmus gibt mit

$$\begin{aligned} \text{Input} &= (n_1, \ldots, n_k) \in N^k \\ \text{Output} &= f(n_1, \ldots, n_k) \in N \end{aligned}$$

bzw. ohne Stop bei $f(n_1, \ldots, n_k) = \text{undef}$

Beispiel 1

Der Algorithmus

INPUT(n);
REPEAT UNTIL FALSE;

„berechnet“ die total undefinierte Funktion $\Omega : n \mapsto \textit{undef.}$

Pi

$\pi = 3.14159\ 26535\ 89793\ 23846\ 26433\ 83279\ 50288\ 41971\ 69399\ 37510\ 58209\ 74944\ 59230$
$78164\ 06286\ 20899\ 86280\ 34825\ 34211\ 70679\ 82148\ 08651\ 32823\ 06647\ 09384\ 46095\ 50582$
$23172\ 53594\ 08128\ 48111\ 74502\ 84102\ 70193\ 85211\ 05559\ 64462\ 29489\ 54930\ 38196\ 44288$
$10975\ 66593\ 34461\ 28475\ 64823\ 37867\ 83165\ 27120\ 19091\ 45648\ 56692\ 34603\ 48610\ 45432$
$66482\ 13393\ 60726\ 02491\ 41273\ 72458\ 70066\ 06315\ 58817\ 48815\ 20920\ 96282\ 92540\ 91715$
$36436\ 78925\ 90360\ 01133\ 05305\ 48820\ 46652\ 13841\ 46951\ 94151\ 16094\ 33057\ 27036\ 57595$
$91953\ 09218\ 61173\ 81932\ 61179\ 31051\ 18548\ 07446\ 23799\ 62749\ 56735\ 18857\ 52724\ 89122$
$79381\ 83011\ 94912\ 98336\ 73362\ 44065\ 66430\ 86021\ 39494\ 63952\ 24737\ 19070\ 21798\ 60943$
$70277\ 05392\ 17176\ 29317\ 67523\ 84674\ 81846\ 76694\ 05132\ 00056\ 81271\ 45263\ 56082\ 77857$
$71342\ 75778\ 96091\ 73637\ 17872\ 14684\ 40901\ 22495\ 34301\ 46549\ 58537\ 10507\ 92279\ 68925$
$89235\ 42019\ 95611\ 21290\ 21960\ 86403\ 44181\ 59813\ 62977\ 47713\ 09960\ 51870\ 72113\ 49999$
$99837\ 29780\ 49951\ 05973\ 17328\ 16096\ 31859\ 50244\ 59455\ 34690\ ...$

Beispiel 2

Definition (Anfangsabschnittsfunktions)

$$f_r(n) = \begin{cases} 1 & \text{falls } n \text{ ein Anfangsabschnitt der} \\ & \quad \text{Dezimalbruchentwicklung von } r \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

Die Funktion $f_\pi(n)$ ist berechenbar.
Es gibt Näherungsverfahren für die Zahl π.

$$f_\pi(3) = 1$$
$$f_\pi(7) = 0$$
$$f_\pi(314) = 1$$
$$f_\pi(31415926535897932384626433832 79) = 1$$

Beispiel 3

Die Funktion

$$g(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der} \\ & \quad \text{Dezimalbruchentwicklung von } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

ist möglicherweise nicht berechenbar.

Unser bisheriges Wissen über die Zahl π reicht nicht aus, um eine Entscheidung über Berechenbarkeit oder Nicht-Berechenbarkeit zu treffen.

Es wird vermutet, π sei eine *absolut normale* Zahl.

Definition (Absolut normale Zahlen)

Eine reelle Zahl x heisst *absolut normal*, wenn für jedes $b \in N$ und jede b-näre Ziffernfolge der Länge k gilt, dass der Anteil ihres Vorkommens in der b-nären Entwicklung von x der Länge n gegen b^{-k} strebt für $n \to \infty$.

Beispiele

- 0.12345678910111213141516... ist normal zur Basis 10.
- Fast alle reellen Zahlen (im Sinne von Lebesgue) sind absolut normal.
- Wenn π normal zur Basis 2 ist, dann gilt auch dass die Kreiszahl alle bisher und zukünftig geschriebenen Bücher irgendwo in codierter Binär-Form enthalten muss (aus Wikipedia).

Beispiel 3 — continued

$$g(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der} \\ & \quad \text{Dezimalbruchentwicklung von } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Es wird vermutet, π sei eine absolut normale Zahl.

Wenn das für π bewiesen ist, gilt $g(n) = 1$ für alle $n \in N$ und diese Funktion ist sicher berechenbar.

Beispiel 4

Ist die Funktion

$$h(n) = g(\overbrace{777...777}^{n-mal})$$

berechenbar?

Antwort: Ja! Denn entweder ist

$$h(n) \equiv 1$$

oder es gibt ein $K \in N$ mit

$$h(n) = \begin{cases} 1 & \text{für } n \leq K \\ 0 & \text{sonst} \end{cases}$$

Beide sind berechenbar.

Beispiel 4

Achtung!

- Berechenbarkeit ist *nicht konstruktiv*.
- Es reicht die Existenz eines Algorithmus'.

$$i(n) = \begin{cases} 1 & \text{falls das LBA-Problem eine positive Lösung hat} \\ 0 & \text{sonst.} \end{cases}$$

Unsere momentane Unwissenheit über den Status des LBA-Problems ändert nichts daran, dass i berechenbar ist, denn i ist entweder die konstante 1-Funktion oder die konstante 0-Funktion, und beide sind berechenbar.

Ist f_r berechenbar?

- f_π ist berechenbar.
- f_e ist berechenbar ($e = 2.71...$ ist die Eulerzahl).
- $f_{\sqrt{2}}$ ist berechenbar.
- Frage Ist f_r für alle r, reell, berechenbar?
- Antwort: Nein! Denn es gibt überabzählbar viele reelle Zahlen und nur abzählbar viele Algorithmen (warum?).
- Es gibt also berechenbare und nicht berechenbare reelle Zahlen.
- Alle rationalen Zahlen sind berechenbar.

Ansätze zur formalen Definition von Berechenbarkeit

Berechenbarkeit kann formal definiert werden durch:[1]

- Turingmaschinen
- WHILE-Programme
- GOTO-Programme
- μ-rekursive Funktionen

Alle Definitionen sind nachweisbar äquivalent!

Ein noch umfassenderer Berechenbarkeitsbegriff wurde nie gefunden. Man ist heute davon überzeugt, damit genau *den* Berechenbarkeitsbegriff erfasst zu haben. Das heißt, wenn von einer Funktion nachgewiesen ist, dass sie *nicht* Turingmaschinen-berechenbar ist, dann folgt, dass die Funktion *überhaupt nicht* berechenbar ist.

[1] Erste Definitionen von Turing und Church 1936.

Zusammenfassung als Churchsche These

Churchsche These.

Die durch die formale Definition der *Turing-Berechenbarkeit* (äquivalent: *WHILE-Berechenbarkeit, GOTO-Berechenbarkeit, μ-Rekursivität*) erfasste Klasse von Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

Syntaktische Komponenten

- *Variablen:* x_0 x_1 x_2 ...
- *Konstanten:* 0 1 2 ...
- *Trennsymbole:* ; :=
- *Operationszeichen:* + −
- *Schlüsselwörter:* LOOP DO END

LOOP Programme: Induktive Definition der Syntax

- Jede Wertzuweisung der Form

$$x_i := x_j + c \text{ bzw. } x_i := x_j - c$$

  ist ein LOOP-Programm (wobei $c \in N$ eine Konstante ist).
- Falls P_1 und P_2 bereits LOOP-Programme sind, dann auch

$$P_1; P_2$$

- Falls P ein LOOP-Programm ist und x_i eine Variable, dann ist auch

$$\text{LOOP } x_i \text{ DO } P \text{ END}$$

  ein LOOP-Programm.

LOOP Programme: Semantik 1

- Ein LOOP-Programm soll k-stellige Funktion $f(n_1, \ldots, n_k)$ berechnen.
- Startwerte $n_1, \ldots, n_k \in N$ gespeichert in Variablen $x_1, \ldots, x_k$.
- Alle anderen Variablen $x_0, x_{k+1}, x_{k+2}, \ldots$ haben Anfangswert 0.
- Wertzuweisung $x_i := x_j + c$ wird wie üblich interpretiert (c ist Konstante)
- Bei $x_i := x_j - c$ wird die *modifizierte Subtraktion* verwendet, also falls $c > x_j$, so wird das Resultat auf 0 gesetzt.

LOOP Programme: Semantik 2

- Ein LOOP-Programm der Form $P_1; P_2$ führt erst P_1 und dann P_2 aus.
- Ein LOOP-Programm der Form LOOP x_i DO P END führt das Programm P sooft aus, wie der Wert der Variablen x_i *zu Beginn* angibt.
- Ändern des Variablenwerts von x_i im Inneren von P hat also keinen Einfluss auf die Anzahl der Wiederholungen.
- Das *Resultat* der Berechnung eines LOOP-Programms (nach Ausführung des LOOP-Programms) ergibt sich als Wert der Variablen x_0.

LOOP-Berechenbarkeit

Eine Funktion $f : N^k \longrightarrow N$ heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f in dem Sinne berechnet, dass P, gestartet mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0.

Bemerkungen:

- LOOP-Programme kennen keine unendlichen Schleifen.
- Alle LOOP-berechenbaren Funktionen sind also *totale* Funktionen.
- Sind alle totalen und intuitiv berechenbaren Funktionen bereits LOOP-berechenbar?
- Antwort: Nein! Gegenbeispiel ist die *Ackermannfunktion*.

Simulationen oder Macros

$x_i := x_j$

$x_i := x_j + c$ mit $c = 0$

$x_i := c$

$x_i := x_k + c$ mit $x_k = 0$, d.h., x_k noch unbenutzte Variable.

IF $x = 0$ THEN A END

$y := 1;$
LOOP x DO $y := 0$ END;
LOOP y DO A END

IF ... THEN A ELSE B END usw.

entsprechend

Addition und Multiplikation

$x_0 := x_1 + x_2$

$x_0 := x_1;$
LOOP x_2 DO $x_0 := x_0 + 1$ END

$x_i := x_j + x_k$

analog

$x_0 := x_1 * x_2$

$x_0 := 0;$
LOOP x_2 DO $x_0 := x_0 + x_1$ END

Das sind zwei geschachtelte LOOP Schleifen!

DIV und MOD

(x_1 DIV x_2) mit $x_2 > 0$: ganzzahliger Anteil beim Teilen

$x_0 := 0;$
LOOP x_1 DO
    IF $x_1 + 1 - x_2 \neq 0$ DO
        $x_1 := x_1 - x_2;$
        $x_0 := x_0 + 1$
    END
END

(x_1 MOD x_2) mit $x_2 > 0$: modulo Funktion

$$x_0 := x_1 - (x_1 \text{DIV} x_2) * x_2$$

Beispielanwendung: $x := (y \text{ DIV } z) + (x \text{ MOD } 5) * y$

WHILE-Schleife als neues Konstrukt

- Falls P ein WHILE-Programm ist und x_i eine Variable, dann ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

  ein WHILE-Programm.
- *Semantik* der WHILE-Schleife: P wird solange wiederholt ausgeführt, wie der Wert von x_i ungleich Null ist.
- Syntax sonst wie bei LOOP-Programmen

Schleife LOOP x DO P END wird überflüssig

$$y := x;$$
$$\text{WHILE } y \neq 0 \text{ DO } y := y - 1;\ P \text{ END}$$

Berechenbarkeit durch WHILE-Programme

Eine Funktion $f : N^k \longrightarrow N$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das f in dem Sinne berechnet, dass P, gestartet mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0 – sofern $f(n_1, \ldots, n_k)$ definiert ist, ansonsten stoppt P nicht.

Turingmaschinen können WHILE-Programme simulieren

- Turingmaschinen können Wertzuweisungen, Sequenzenbidung, WHILE-Schleifen realisieren
- i-te Variable x_i auf i-tem Band
- Einband-Maschinen simulieren Mehrbandmaschinen

Satz

Turingmaschinen können WHILE-Programme simulieren. Das heißt, jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

- Umkehrung über Umweg über GOTO-Programme.

GOTO-Programme

- Sequenzen von Anweisungen A_i mit Marken M_i
- $M_1 : A_1;\ M_2 : A_2;\ \ldots\ ;\ M_k : A_k$
- Als mögliche Anweisungen A_i sind zugelassen:
  Wertzuweisungen: $x_i := x_j \pm c$
  unbedingter Sprung: GOTO M_i
  bedingter Sprung: IF $x_i = c$ THEN GOTO M_j
  Stopanweisung: HALT
- Überflüssige Marken kann man weglassen
- Semantik klar
- GOTO-Berechenkeit definiert wie WHILE-Berechenbarkeit

Simulation von WHILE-Programmen durch GOTO-Programme

WHILE $x_i \neq 0$ DO P END wird simuliert durch

$$
\begin{aligned}
M_1 :\ & \text{IF } x_i = 0 \text{ THEN GOTO } M_2; \\
& P; \\
& \text{GOTO } M_1 \ ; \\
M_2 : \ & \ldots
\end{aligned}
$$

Satz

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.
Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

Simulation von $M_1 : A_1;\ M_2 : A_2;\ \ldots\ ; M_k : A_k$

$$
\begin{array}{l}
count := 1; \\
\text{WHILE } count \neq 0 \text{ DO} \\
\quad \text{IF } count = 1 \text{ THEN } A'_1 \text{ END}; \\
\quad \text{IF } count = 2 \text{ THEN } A'_2 \text{ END}; \\
\quad \vdots \\
\quad \text{IF } count = k \text{ THEN } A'_k \text{ END} \\
\text{END}
\end{array}
$$

$$
A'_i = \begin{cases}
x_j := x_l \pm c;\ count := count + 1 & \text{falls } A_i = x_j := x_l \pm c \\
count := n & \text{falls } A_i = \text{GOTO } M_n \\
\text{IF } x_j = c \text{ THEN } count := n \\
\text{ELSE } count := count + 1 \text{ END} & \text{falls } A_i = \begin{array}{l}\text{IF } x_j = c \\ \text{THEN GOTO } M_n\end{array} \\
count := 0 & \text{falls } A_i = \text{HALT}
\end{cases}
$$

Kleensche Normalform

Satz

Jedes GOTO-Programm kann durch ein WHILE-Programm (mit nur einer WHILE-Schleife) simuliert werden. Also ist jede GOTO-berechenbare Funktion auch WHILE-berechenbar.

Satz Kleensche Normalform

Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.

Beweis: Sei ein beliebiges WHILE-Programm P zur Berechnung einer Funktion f gegeben. Wir formen P zunächst um in ein äquivalentes GOTO-Programm P' und dann wieder zurück in ein äquivalentes WHILE-Programm P''. Dieses hat nur noch eine WHILE-Schleife.

Komplexitätstheorie – Worum geht es?

- Klassifikation algorithmischer Probleme (= entscheidbare Sprachen) nach Bedarf an Resourcen (Rechenzeit, Speicher) in Abhängigkeit der Eingabelänge n.
  - Obere Schranken $O(f(n))$. *Ein* Algorithmus liefert eine solche.
  - Untere Schranken $\Omega(f(n))$. Schwieriger, da *alle* Algorithmen berücksichtigt werden müssen, auch die noch unbekannten.
- Stärke verschiedener Maschinenmodelle, insbesondere deterministische versus nicht-deterministische.
- Hier: Zentrale offene Frage: $P = NP$ oder $P \neq NP$. Dazu Strukturtheorie von Cook (1971), Karp (1972).

Definition ($TIME(f(n))$)

Zu $f : N \longrightarrow N$ enthalte die Klasse $TIME(f(n))$ all die Sprachen A, für die es eine deterministische Mehrband-Turingmaschine M gibt mit $A = T(M)$ und $time_M(x) \leq f(|x|)$.

Hierbei bedeutet $time_M : \Sigma^* \longrightarrow N$ die Anzahl der Rechenschritte von M bei Eingabe x.

- Mehrbandmaschine liefert realistischere Funktionen f. (Einbandmaschine braucht Zeit zum Positionieren des Bandes, die nicht das Problem widerspiegelt.)
- Mehrbandmaschinen (mit k Bändern) können durch Einbandmaschinen (mit $2k$ "Spuren") simuliert werden.
- Analyse: $O(f(n))$ bei Mehrbandmaschine ergibt bei Einbandmaschine $O(f(n)^2)$.

Die Komplexitätsklasse P

Betrachte Funktionsklasse der Polynome $p : N \longrightarrow N$.

- $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \; ; \; a_i, k \in N$
- $p(n)$ ist folglich monoton wachsend.
- $time_M(x)$ für Mehrbandmaschine M polynomial $\Rightarrow$ $time_{M'}(x)$ für simulierende Einbandmaschine M' polynomial.

Definition (Komplexitätsklasse P)

Die Komplexitätsklasse P ist die Klasse der Sprachen A, für die es eine Turingmaschine M und ein Polynom p gibt mit $T(M) = A$ und $time_M(x) \leq p(|x|)$. Also

$$\mathrm{P} = \bigcup_{\text{p Polynom}} \mathrm{TIME(p(n))}$$

Effiziente Algorithmen

- Nachweis polynomialer Komplexität für einen Algorithmus:
  Zeige Komplexität $O(n^k)$ für eine Konstante k.
- Ein Algorithmus mit Komplexität $n \log n$ ist auch als polynomial (obwohl $n \log n$ kein Polynom ist), denn $n \log n = O(n^2)$.
- Funktionen wie $n^{\log n}$ oder 2^n können dagegen durch kein Polynom nach oben beschränkt werden.
- Man kann argumentieren, dass die Klasse P genau diejenigen Probleme umfasst, für die *effiziente* Algorithmen existieren. Ein Algorithmus z.B. der Komplexität 2^n ist dagegen nicht effizient.

Sprachen $A \in \mathrm{P}$ sind LOOP-berechenbar

Die Klasse P, ebenso wie noch weit größere Komplexitätsklassen, etwa $TIME(2^n)$ oder gar $TIME(\underbrace{2^{2^{\cdots^{2^2}}}}_{n\text{-mal}})$ sind in der Klasse der primitiv rekursiven bzw. LOOP-berechenbaren Sprachen enthalten.

Beweisskizze:

- Sei $A \in TIME(f(n))$ und M eine TM mit $A = T(M)$ und $time_M(x) \leq f(n), n = |x|$.
- Simuliere TM M durch GOTO Programm.
- Simuliere GOTO- durch WHILE-Programm (1 Schleife)
- Es gibt maximal $f(n)$ Schleifendurchläufe.
- Ersetze "WHILE count $\neq 0$ DO"
  durch "$y := f(n)$; LOOP y DO"
- Bemerkung: $f(n)$ muss dazu LOOP-berechenbar sein.

Uniformes versus logarithmisches Kostenmaß

- Die Klasse P ist unabhängig von der Maschine oder Programmiersprache, relativ zu der *time* definiert wird.
- Definition auch mit WHILE-Programmen möglich, wenn das *logarithmische Kostenmaß* verwendet wird. Beispiel dazu:

$$\begin{array}{l}\text{INPUT } (w);\\ n := |w|;\ x := 2;\\ \text{LOOP } n \text{ DO } x := x * x \text{ END};\\ \text{OUTPUT } (x)\end{array}$$

- Anzahl der Operationen ist $O(n)$, also polynomial (bei *uniformen Kostenmaß*, d.h. 1 Multiplikation= 1 Op).
- Aber: Programm berechnet $2^{(2^n)}$ und eine TM braucht schon 2^n Schritte zur Ausgabe des Ergebnisses.

Definition (Komplexitätsklasse NP)

Für nichtdeterministische Turingmaschinen M sei

$$ntime_M(x) = \begin{cases} \text{min [Länge einer akzeptierenden} \\ \quad \text{Rechnung von } M \text{ auf } x], & x \in T(M) \\ 0, & x \notin T(M) \end{cases}$$

Für $f : N \longrightarrow N$ bestehe die Klasse $NTIME(f(n))$ aus allen Sprachen A, für die es eine *nichtdeterministische* Mehrband-Turingmaschine M gibt mit $A = T(M)$ und $ntime_M(x) \leq f(|x|)$. Ferner definieren wir:

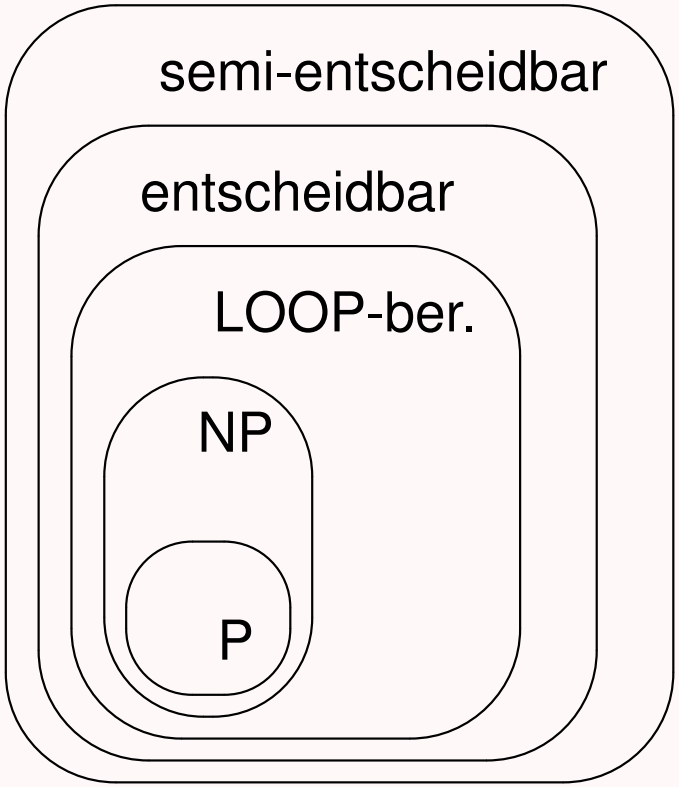
$$\text{NP} = \bigcup_{p \text{ Polynom}} NTIME(p(n))$$

Problem (P-NP-Problem (1970))

$P \subseteq NP$ *ist offensichtlich.*
Das P-NP-Problem ist die offene Frage, ob $P = NP$ *gilt.*
Man vermutet $P \neq NP$*.*

- Dies wird oft als die wichtigste Frage der theoretischen Informatik überhaupt angesehen.
- Viele wichtigen Aufgaben liegen in NP (z.B. Travelling Salesman, Erfüllbarkeitsproblem der Aussagenlogik, usw.)
- Polynomiale Algorithmen für diese Sprachen sind aber nicht bekannt.
- Strukturtheorie für das P-NP-Problem: zentral dabei die sog. *NP-vollständigen* Probleme.
- *Alle* diese Probleme besitzen polynomiale Algorithmen (nämlich falls $P = NP$) – oder keines (falls $P \neq NP$).

Überblick



- In NP, nicht in P:
  Travelling Salesman
  (wenn P $\neq$ NP).
- LOOP-berechenbar, nicht in NP:
  $n!$.
- Entscheidbar, nicht
  LOOP-berechenbar: Ackermann
  Funktion.
- Semi-entscheidbar, nicht
  entscheidbar:
  echte Typ-0 Sprache.

Definition (Polynomiale Reduktion)

Seien $A \subseteq \Sigma^*$, $B \subseteq \Gamma^*$ Sprachen.
A heißt auf B *polynomial reduzierbar*, symbolisch $A \leq_p B$, falls eine totale, berechenbare Funktion polynomialer Komplexität $f : \Sigma^* \to \Gamma^*$ existiert, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \Leftrightarrow f(x) \in B$

Lemma

Falls $A \leq_p B$ und $B \in P$ (bzw. $B \in NP$),
so ist auch $A \in P$ (bzw. $A \in NP$).

Lemma

Falls $A \leq_p B$ und $B \in P$ (bzw. $B \in NP$),
so ist auch $A \in P$ (bzw. $A \in NP$).

Beweis:

- Es gelte $A \leq_p B$ mittels Funktion f, die durch TM M_f berechnet wird.
- Das Polynom p begrenzt die Rechenzeit von M_f.
  Ferner sei $B \in P$ mittels TM M.
- Die Rechenzeit von M sei durch ein Polynom q begrenzt.
- Die Rechenzeit der Hintereinanderschaltung $M_f; M$
  ist bei Eingabe x ist durch

$$p(|x|) + q(|f(x)|) \leq p(|x|) + q(p(|x|))$$

  polynomial beschränkt.

Definition (NP-Härte)

Eine Sprache A heißt *NP-hart*, falls für alle Sprachen $L \in NP$ gilt: $L \leq_p A$.

Definition (NP-Vollständigkeit)

Eine Sprache A heißt *NP-vollständig*, falls A *NP-hart* ist und $A \in NP$ gilt.

- *Intuition:* A ist NP-vollständig, falls A mindestens so schwierig ist wie *jedes* Problem in NP.
- $\leq_p$ ist eine *transitive* Relation auf Sprachen: Mit einem bekannten, NP-harten Problem A ist der Nachweis der NP-Härte für ein anderes Problem B evtl. leicht: Man zeige $A \leq_p B$.
  Für alle NP-Probleme folgt dann aus $L \leq_p A$ und $A \leq_p B$ mittels Transitivität, dass auch $L \leq_p B$ gilt.

Satz.

Sei A NP-*vollständig*, dann gilt:

$$A \in P \Leftrightarrow P = NP$$

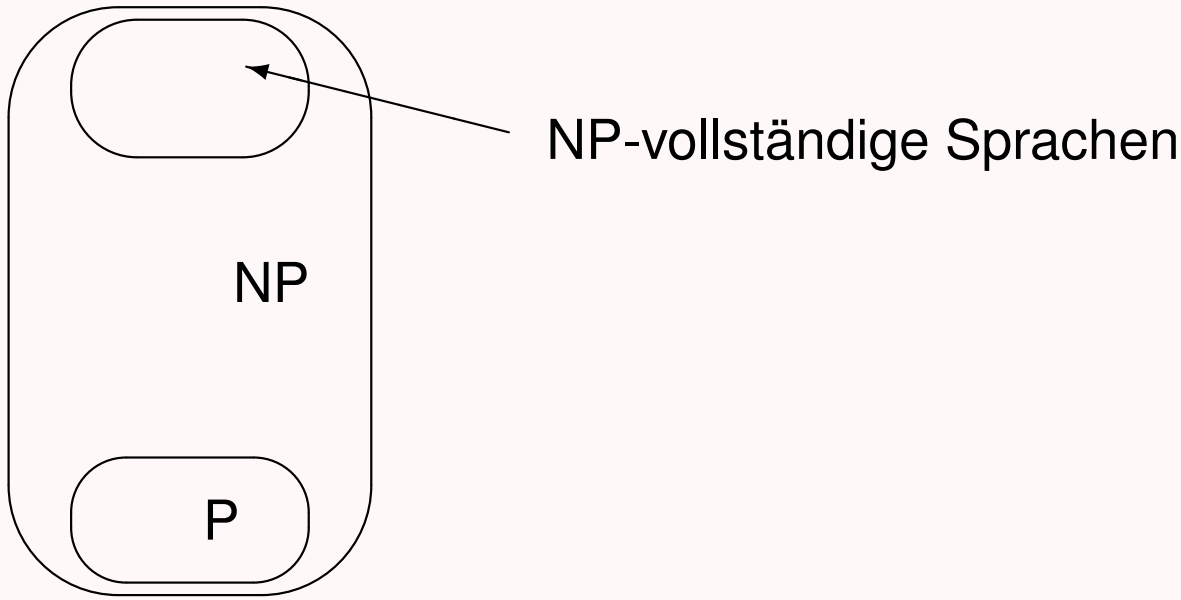
Beweis:

Sei $A \in P$ und L eine beliebige Sprache in NP. Da A NP-hart ist, gilt $L \leq_p A$, somit folgt $L \in P$. Da L beliebig aus NP gewählt war, folgt $P = NP$

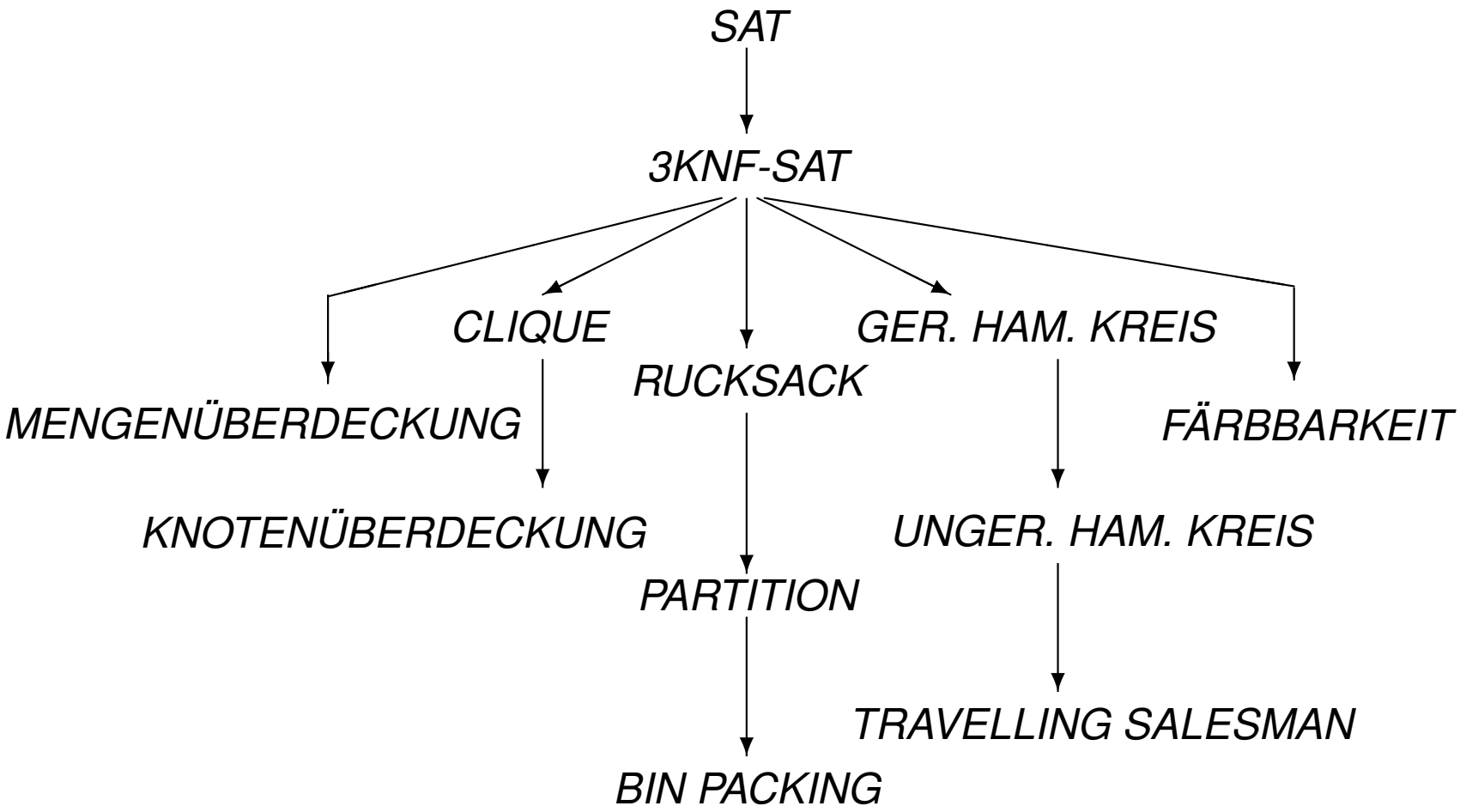
NP-Vollständigkeit

- NP-vollständige Probleme sind in gewissem Sinn die „schwierigsten “ in NP.
- Vermutung: $P \neq NP$

Skizze



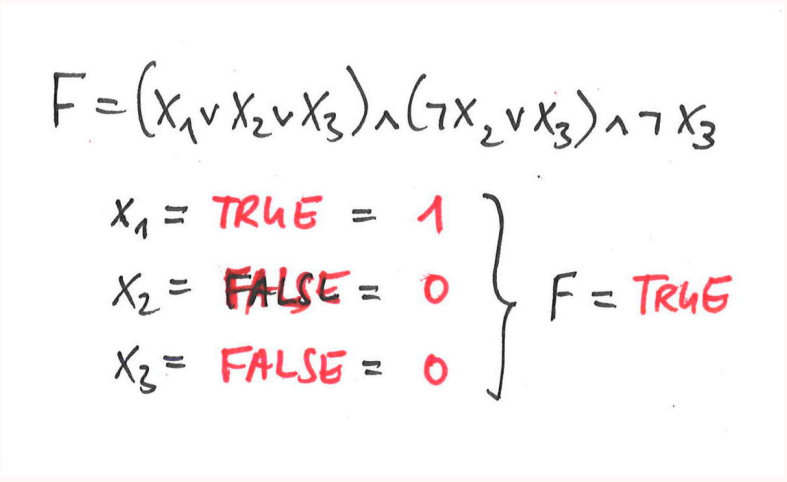
Reduktionsbaum für Ableitung NP-vollständiger Probleme



Problemdefinition

Gegeben: Eine Boolesche Formel F in konjunktiver Normalform (Klauselform) mit höchstens 3 Literalen pro Klausel.

Gefragt: Ist F erfüllbar?



Satz

3KNF-SAT ist NP-vollständig.

Beweis der NP-Vollständigkeit:

Zu zeigen sind zwei Charakteristiken:

- 3KNF-SAT $\in$ NP
- 3KNF-SAT ist NP-schwer

3KNF-SAT $\in$ NP
Guess-and-check Argumentation: Sei M eine Turingmaschine, die zu einer Formel F in 3KNF nichtdeterministisch eine Belegung aller Variablen „rät“ und überprüft, ob F erfüllt ist. Beide Teilaufgaben sind in polynomieller Zeit ausführbar, somit gilt 3KNF-SAT $\in$ NP.

3KNF-SAT ist NP-schwer
Zeige SAT $\leq_p$ 3KNF-SAT.

Beweis der NP-Vollständigkeit:

Reduziere SAT in polynomialzeit erfüllbarkeitsäquivalent auf 3KNF-SAT:

- Wende die de Morganschen Regeln an, um Negationen direkt vor die jeweiligen Literale zu setzen. $\mathcal{O}(|F|^2)$
- Führe zusätzliche Klammern ein, so dass durch Kon- bzw. Disjunktionen nur

  zwei (negierte) Literale
  ein (negiertes) Literal und eine Formel
  zwei Formeln

  verbunden werden. $\mathcal{O}(|F|)$
- Füge für jede Teilformel, die durch Klammerung entstanden ist, eine neue, äquivalente Variable y_i ein. $\mathcal{O}(|F|)$
- Wende die Äquivalenzregeln der Aussagenlogik für Konjunktion und Disjunktion an:

  $$(y_i \leftrightarrow \alpha \wedge \beta) \Leftrightarrow (\neg y_i \vee \alpha) \wedge (\neg y_i \vee \beta) \wedge (\neg\alpha \vee \neg\beta \vee y_i)$$
  $$(y_i \leftrightarrow \alpha \vee \beta) \Leftrightarrow (y_i \vee \neg\alpha) \wedge (y_i \vee \neg\beta) \wedge (\alpha \vee \beta \vee \neg y_i)$$

  Die Laufzeit liegt in $\mathcal{O}(|F|)$, da die Teilformeln konstante Größe besitzen.

Beispielüberführung:

Sei $F = \neg(\neg(x_1 \vee \neg x_3) \vee x_2)$ gegeben.

- Durch Anwendung der de Morganschen Regeln folgt:

$$F_1 = ((x_1 \vee \neg x_3)\neg x_2)$$

- Die Klammerbedingung ist im Beispiel schon erfüllt.
- Füge für die markierten Teilformeln

$$F_1 = \underbrace{(\underbrace{(x_1 \vee \neg x_3)}_{y_1} \wedge \neg x_2)}_{y_0}$$

neue Variablen ein:

$$F_2 = y_0 \wedge (y_0 \leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \leftrightarrow (x_1 \vee \neg x_3))$$

Beispielüberführung:

- Anwendung der Äquivalenzregeln für Konjunktionen und Disjunktion liefert schließlich das Ergebnis:

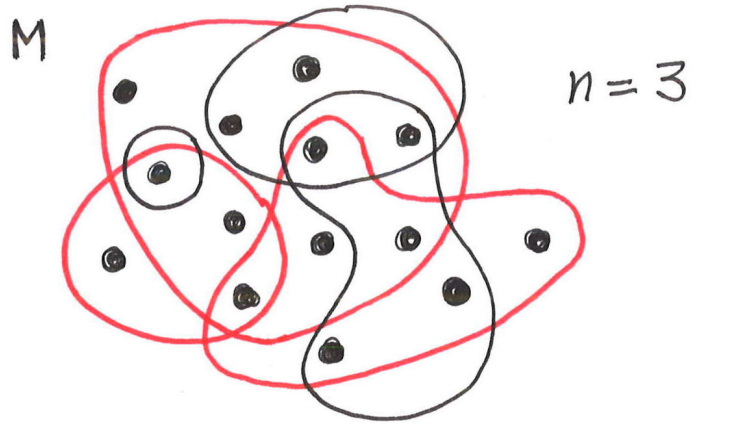
$$F_3 = y_0 \wedge (\neg y_0 \vee y_1) \wedge (\neg y_0 \vee \neg x_2) \wedge (y_0 \vee \neg y_1 \vee x_2) \wedge (y_1 \vee \neg x_1) \\ \wedge (y_1 \vee x_3) \wedge (\neg y_1 \vee x_1 \vee \neg x_3)$$

Somit wurde $F = \neg(\neg(x_1 \vee \neg x_3) \vee x_2)$ erfüllbarkeitsäquivalent in F_3 in 3-KNF überführt. □

Problemdefinition

Gegeben: Ein Mengensystem über einer endlichen Grundmenge M, also $T_1, \ldots, T_k \subseteq M$, sowie eine Zahl $n \leq k$.

Gefragt: Gibt es eine Auswahl aus n Mengen $T_{i_1}, \ldots, T_{i_n}$, in deren Vereinigung alle Elemente aus M liegen?



Satz

Mengenüberdeckung ist NP-vollständig.

Beweis der NP-Vollständigkeit:

Zeige 3KNF-Sat $\leq_p$ MENGENÜBERDECKUNG

Sei $F = K_1 \wedge \ldots \wedge K_m$ eine Formel in 3KNF mit n Variablen und m Klauseln.
Wähle als Grundmenge $M = \{1, 2, \ldots, m, m+1, \ldots, m+n\}$
Für $i = 1, \ldots, n$ sei

$$T_i = \{j | x_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\}$$
$$T_i' = \{j | \bar{x}_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\}$$

Das Mengensystem bestehe aus $T_1, \ldots, T_n, T_1', \ldots, T_n' \subseteq M$

Beweis der NP-Vollständigkeit:

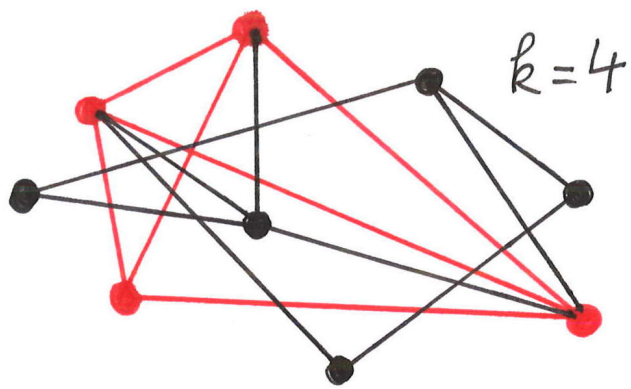
Wenn es eine erfüllende Belegung der n Variablen für F gibt, so können wir eine Teilauswahl des Mengensystems bilden, indem wir für $i = 1, \ldots, n$ gerade die T_i auswählen, wenn in der Belegung $x_i = 1$ gilt, ansonsten wählen wir T_i'. Diese Teilmenge besteht aus n Mengen und hat die Eigenschaft, dass jedes Element aus M darin vorkommt.

Sei umgekehrt $\{U_1, \ldots, U_n\} \subseteq \{T_1, \ldots, T_n, T_1', \ldots, T_n'\}$ eine n-elementige Teilmenge des Mengensystems, die die Grundmenge M umfasst. In dieser Auswahl muss für jedes $i = 1, \ldots, n$ genau eine Menge enthalten sein, die die Zahl $m + i$ enthält, z.B. $U_1 \in \{T_1, T_1'\}, U_2 \in \{T_2, T_2'\}, \ldots, U_n \in \{T_n, T_n'\}$. Die Belegung, die für $i = 1, \ldots, n$ die Variable x_i genau dann auf 1 setzt, wenn $U_i = T_i$, muss eine erfüllende Belegung sein, da in der Mengenauswahl alle Zahlen aus $\{1, \ldots, m\}$ vorkommen. □

CLIQUE

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in N$.

Gefragt: Besitzt G eine „Clique“ der Größe mindestens k? (Dies ist eine Teilmenge V' der Knotenmenge mit $|V'| \geq k$ und für alle $u, v \in V'$ mit $u \neq v$ gilt: $\{u, v\} \in E$).



Satz

CLIQUE ist NP-vollständig.

Beweis der NP-Vollständigkeit:

Zeige 3KNF-SAT $\leq_p$ CLIQUE

Sei $F = (z_{11} \vee z_{12} \vee z_{13}) \wedge \ldots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$, wobei
$z_{ij} \in \{x_1, x_2, \ldots\} \cup \{\neg x_1, \neg x_2, \ldots\}$
F wird nun ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$ zugeordnet.
Setze:

$$V = \{(1,1), (1,2), (1,3), \ldots, (m,1), (m,2), (m,3)\}$$

$$E = \{\{(i,j), (p,q)\} | i \neq p \text{ und } z_{ij} \neq \not{z}_{pq}\}$$

$$k = m$$

Beweis der NP-Vollständigkeit:

Für G wurde

$$V = \{(1,1),(1,2),(1,3),\ldots,(m,1),(m,2),(m,3)\}$$
$$E = \{\{(i,j),(p,q)\} | i \neq p \text{ und } z_{ij} \neq \neg z_{pq}\}$$
$$k = m$$

gesetzt.

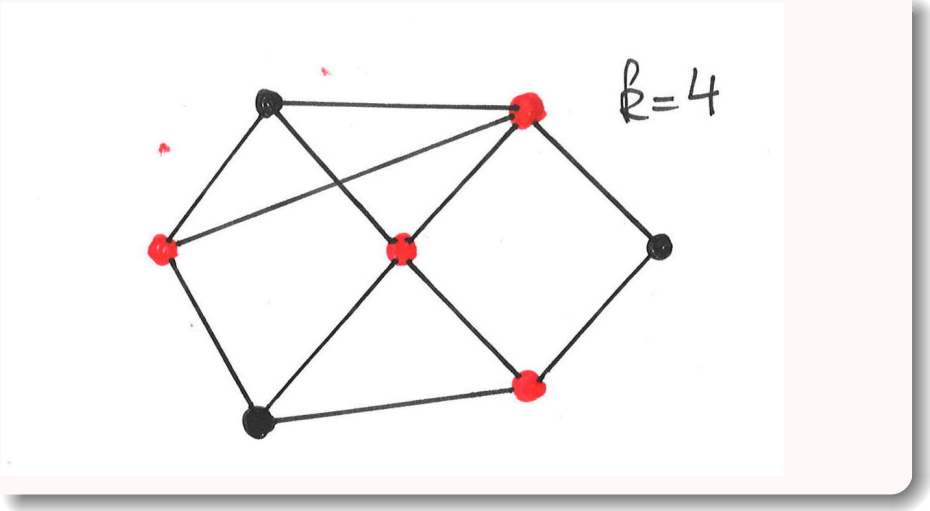
Nun gilt:
F ist erfüllbar durch eine Belegung B
$\Leftrightarrow$ In jeder Klausel nimmt ein Literal unter B den Wert 1 an,
z.B. $z_{1,j_1}, z_{2,j_2}, \ldots, z_{m,j_m}$
$\Leftrightarrow$ Es gibt Literale $z_{1,j_1}, \ldots, z_{m,j_m}$, die paarweise nicht komplementär sind.
$\Leftrightarrow$ Es gibt Knoten $(1,j_1),\ldots,(m,j_m)$ in G, die paarweise verbunden sind.
$\Leftrightarrow$ G hat eine Clique der Größe k.

Die Abbildung $F \mapsto (G,k)$ ist offensichtlich in Polynomialzeit berechenbar. □

KNOTENÜBERDECKUNG

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in N$.

Gefragt: Besitzt G eine „überdeckende Knotenmenge“ der Größe höchstens k?
(Dies ist eine Teilmenge $V' \subseteq V$ mit $|V'| \leq k$, so dass für alle Kanten $\{u, v\} \in E$ gilt: $u \in V'$ oder $v \in V'$).



Satz

KNOTENÜBERDECKUNG ist NP-vollständig.

Beweis der NP-Vollständigkeit

Zeige CLIQUE $\leq_p$ KNOTENÜBERDECKUNG

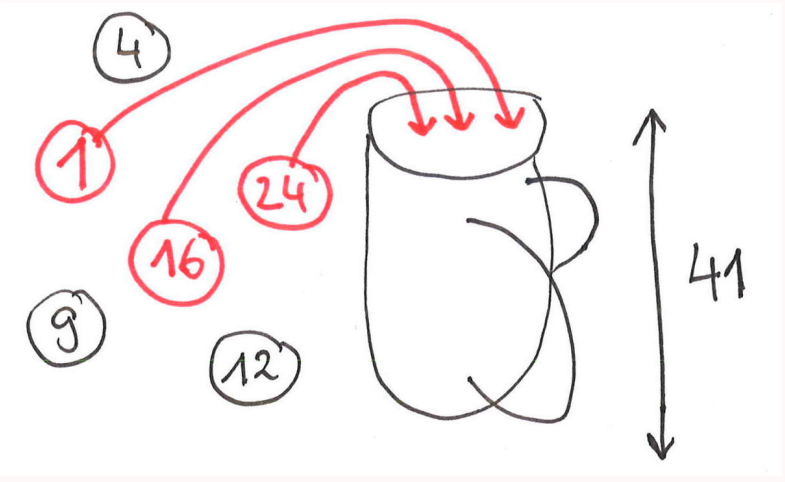
Die Reduktion ist trivial:
Bilde den Graphen $G = (V, E)$ auf seinen Komplementgraphen

$$\overline{G} = (V, \{\{u, v\} | u, v \in V \{u, v\} \notin E\})$$

und die Zahl k auf $|V| - k$ ab. Diese Abbildung ist offensichtlich polynomiell.
□

Gegeben: Natürliche Zahlen $a_1, a_2, \ldots, a_k \in N$ und $b \in N$.

Gefragt: Gibt es eine Teilmenge $I \subseteq \{1, 2, \ldots, k\}$ mit $\sum_{i \in I} a_i = b$?



Satz

RUCKSACK ist NP-vollständig.

Beweis der NP-Vollständigkeit:

- Reduziere *3KNF-SAT* auf *RUCKSACK*.
- Sei

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge \cdots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

  eine Formel in 3-konjunktiver Normalform, wobei

$$z_{ij} \in \{x_1, x_2, \ldots, x_n\} \cup \{\neg x_1, \neg x_2, \ldots, \neg x_n\}$$

- Also ist m die Anzahl der Klauseln und n die Anzahl der vorkommenden Variablen.
- Gemäß dem *RUCKSACK*-Problem sind nun die Zahlen $a_1, ..., a_k$ und b anzugeben.

Beweis der NP-Vollständigkeit:

- Die Zahl b ist gegeben durch:

$$b = \underbrace{444\ldots44}_{m}\underbrace{11\ldots11}_{n}$$

(im Dezimalsystem: m mal die Ziffer 4, dann n mal die Ziffer 1)

- Zur einfacheren Darstellung Fixierung auf 3 Klauseln und 5 vorkommende Variablen. Dann lautet b also:

$$b = 444\,11111$$

Außerdem Fixierung auf ein konkretes Beispiel. Sei also

$$F = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (\neg x_2 \vee \neg x_2 \vee \neg x_5)$$

Beweis:

- Unterteile die Zahlen a_i in verschiedene Klassen.
- Seien die Zahlen $v_1, \ldots, v_n$ folgendermaßen definiert:

  An der i-ten Position in v_1 steht eine 1, falls die Variable x_1 in Klausel i – in positiver Form – vorkommt
  (eine 2 falls die Variable zweimal vorkommt, eine 3 falls sie dreimal vorkommt).
  Im hinteren Ziffernblock steht an Position 1 eine 1.

  Analog ist v_2, bezogen auf die Vorkommen von x_2, definiert. Ferner steht im hinteren Ziffernblock die 1 an Position 2, etc.

Beweis:

- Konkret für das obige Beispiel lauten die Zahlen:

$$
\begin{aligned}
v_1 &= 100\ 10000 \\
v_2 &= 000\ 01000 \\
v_3 &= 000\ 00100 \\
v_4 &= 010\ 00010 \\
v_5 &= 110\ 00001
\end{aligned}
$$

Beweis:

- Die Zahlen v_i' sind völlig analog definiert,
  nur auf die *negativen Vorkommen* der Variablen bezogen.
- Am konkreten Beispiel:

$$\begin{aligned} v_1' &= 010\ 10000 \\ v_2' &= 002\ 01000 \\ v_3' &= 100\ 00100 \\ v_4' &= 000\ 00010 \\ v_5' &= 001\ 00001 \end{aligned}$$

Beweis:

- Schließlich noch die Zahlen c_j und d_j für $j = 1, \ldots, m$, die so aufgebaut sind, dass im ersten Ziffernblock an Position j eine 1 (bei c_j), bzw. eine 2 (bei d_j) steht.
- Konkret:

$$
\begin{aligned}
c_1 &= 100\ 00000 \\
c_2 &= 010\ 00000 \\
c_3 &= 001\ 00000 \\
\\
d_1 &= 200\ 00000 \\
d_2 &= 020\ 00000 \\
d_3 &= 002\ 00000
\end{aligned}
$$

Beweis:

- Damit folgt: Wenn Formel F eine erfüllende Belegung besitzt, so lässt sich eine Auswahl aus den Zahlen treffen, die sich zu b aufsummiert:
  Für jedes $i = 1, \ldots, n$ nehme man v_i bzw. v_i' in die Auswahl auf, falls die Belegung die Variable x_i auf 1 bzw. auf 0 setzt.
- Konkretes Beispiel: Eine erfüllende Belegung wäre

$$\begin{aligned} x_1 &\mapsto 1 \\ x_2 &\mapsto 0 \\ x_3 &\mapsto 0 \\ x_4 &\mapsto 1 \\ x_5 &\mapsto 0 \end{aligned}$$

Beweis:

- Dies ergibt die Auswahl $v_1, v_2', v_3', v_4, v_5'$.
- Diese Zahlen summieren sich auf zu 21311111.
- Man nehme nun noch geeignet c_j und/oder d_j hinzu,
  in diesem Fall d_1, c_2, d_2, c_3,
  so erhält man die gewünschte Summe b $(= 444\ 11111)$.

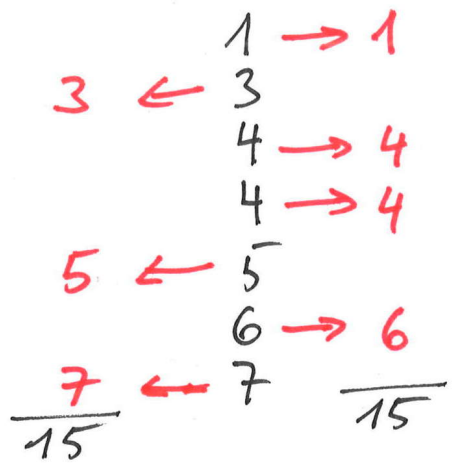
Beweis:

- Sei umgekehrt eine Auswahl von Zahlen gegeben, die sich exakt zu b aufsummiert.
  Da die Konstruktion der Zahlen so angelegt ist, dass keine Überträge zwischen den einzelnen Ziffern möglich sind, muss diese Auswahl derart sein, dass für jedes i *genau eine* der beiden Zahlen v_i und v'_i ausgewählt wurde.
- Nun lässt sich behaupten, dass die Belegung, die man umgekehrt dieser Auswahl von v_i, v'_i zuordnen kann, eine erfüllende Belegung für F ist, also in jeder Klausel mindestens ein Literal wahr macht.
- Man nehme an, dies sei nicht so, dann bleibt für irgendein j die j-te Ziffer (die der Klausel j zugeordnet ist) beim Aufsummieren gleich 0.
  Dann kann aber durch Hinzuaddieren von c_j und d_j nicht mehr die in b geforderte Ziffer 4 erreicht werden. □

PARTITION

gegeben: Natürliche Zahlen $a_1, a_2, \ldots, a_k \in N$.

gefragt: Gibt es eine Teilmenge $J \subseteq \{1, 2, \ldots, k\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$?



Satz

PARTITION ist NP-vollständig.

Beweis:

- Sei $(a_1, a_2, \ldots, a_k, b)$ gegebenes *RUCKSACK*-Problem.
- Man setze $M = \sum_{i=1}^{k} a_i$ und definieren folgende Abbildung

$$(a_1, a_2, \ldots, a_k, b) \quad \mapsto \quad (a_1, a_2, \ldots, a_k, M - b + 1, b + 1)$$

- Man zeige nun, dass diese Abbildung
  eine Reduktion von *RUCKSACK* nach *PARTITION* vermittelt.

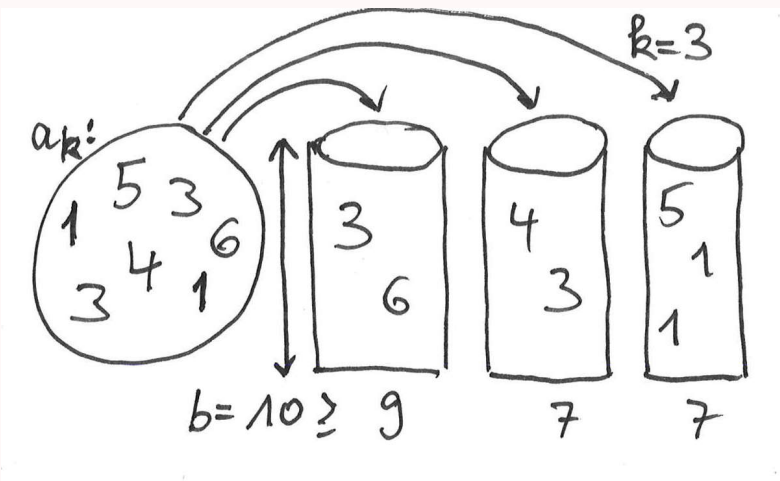
Beweis:

- Wenn $I \subseteq \{1, \ldots, k\}$ eine Lösung des RUCKSACK-Problems darstellt, dann ist $I \cup \{k+1\}$ eine Lösung des Partitionsproblems.
- Wenn J eine Lösung des Partitionsproblems ist, dann können die Zahlen $M - b + 1$ und $b + 1$ nicht beide in J oder beide nicht in J liegen, da ihre Summe zu groß ist.
- o.B.d.A. liege $M - b + 1$ in J, dann ist J (ohne die Zahl $M - b + 1$) eine Lösung des RUCKSACK-Problems, denn die Summe ist gerade b. □

BIN PACKING

gegeben: Eine „Behältergröße“ $b \in N$, die Anzahl der Behälter $k \in N$, „Objekte“ $a_1, a_2, \ldots, a_n \leq b$.

gefragt: Können die Objekte so auf die k Behälter verteilt werden, dass kein Behälter überläuft? (Das heißt: gefragt ist, ob eine Abbildung $f : \{1, \ldots, n\} \longrightarrow \{1, \ldots, k\}$ existiert, so dass für alle $j = 1, \ldots, k$ gilt: $\sum_{f(i)=j} a_i \leq b$).



Satz

BIN PACKING ist NP-vollständig.

Beweis:

- Man zeige *PARTITION* $\leq_p$ *BIN PACKING*.
- Reduktion wird bewerkstelligt durch:

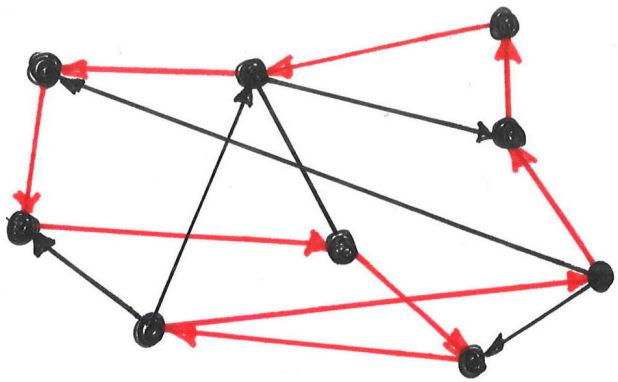
$$(a_1, \ldots, a_k) \quad \mapsto \quad \begin{cases} \text{Behältergröße:} & b = \sum_{i=1}^{k} a_i/2 \\ \text{Zahl der Behälter:} & k = 2 \\ \text{Objekte:} & a_1, \ldots, a_k \end{cases}$$

□

GERICHTETER HAMILTON-KREIS

gegeben: Ein gerichteter Graph $G = (V, E)$.

gefragt: Besitzt G einen Hamilton-Kreis? (Dies ist eine Permutation π der Knotenindizes $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$, so dass für $i = 1, \ldots, n-1$ gilt: $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ und außerdem $(v_{\pi(n)}, v_{\pi(1)}) \in E$).



Satz

GERICHTETER HAMILTON-KREIS ist NP-vollständig.

Beweis:

- Reduziere *3KNF-SAT* auf *GERICHTETER HAMILTON-KREIS*
- Sei

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge \cdots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

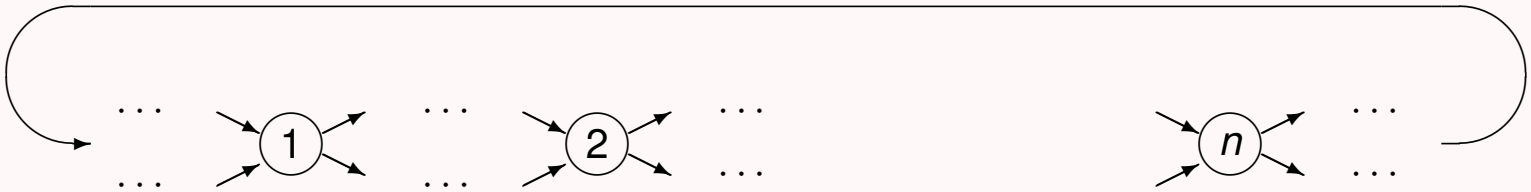
  eine Formel in 3-konjunktiver Normalform, wobei

$$z_{ij} \in \{x_1, x_2, \ldots, x_n\} \cup \{\neg x_1, \neg x_2, \ldots, \neg x_n\}$$

- Also ist m die Anzahl der Klauseln und n die Anzahl der vorkommenden Variablen.

Beweis:

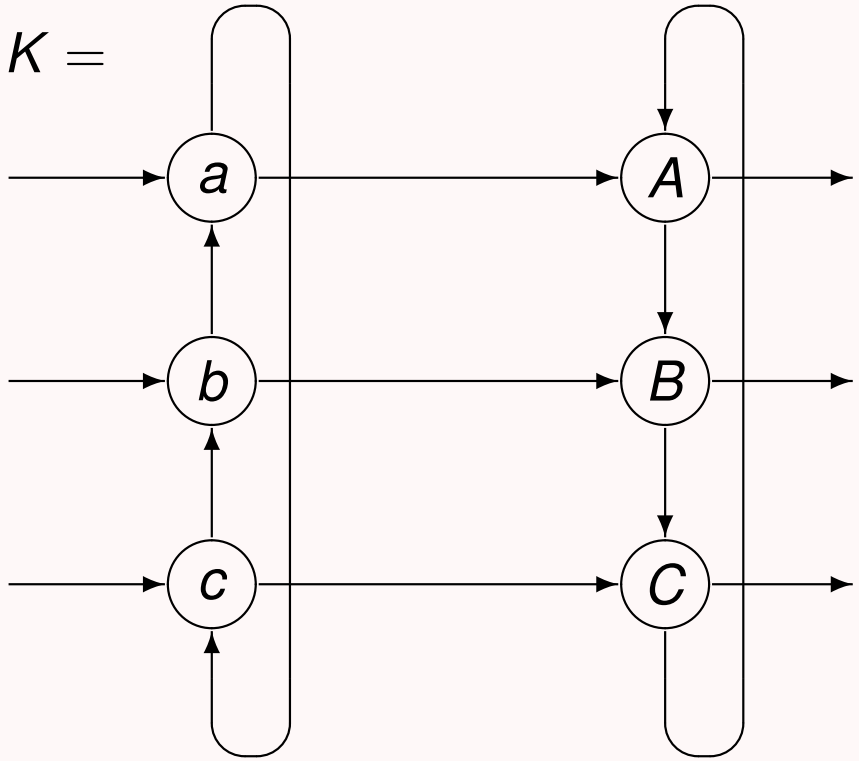
- Konstruieren den gesuchten Graphen mit den Knoten $1, 2, \ldots, n$.



- Von jedem Knoten i gehen jeweils zwei Kanten aus.
- Entsprechenden Pfade führen dann durch weitere Teilgraphen K und enden im Knoten $i + 1$ (vom Knoten n aus führen die Pfade zurück zum Knoten 1).

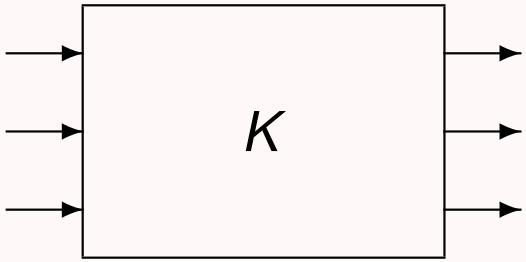
Beweis:

- Vom Graphen K stehen m Kopien $K_1, \ldots, K_m$ bereit.



Beweis:

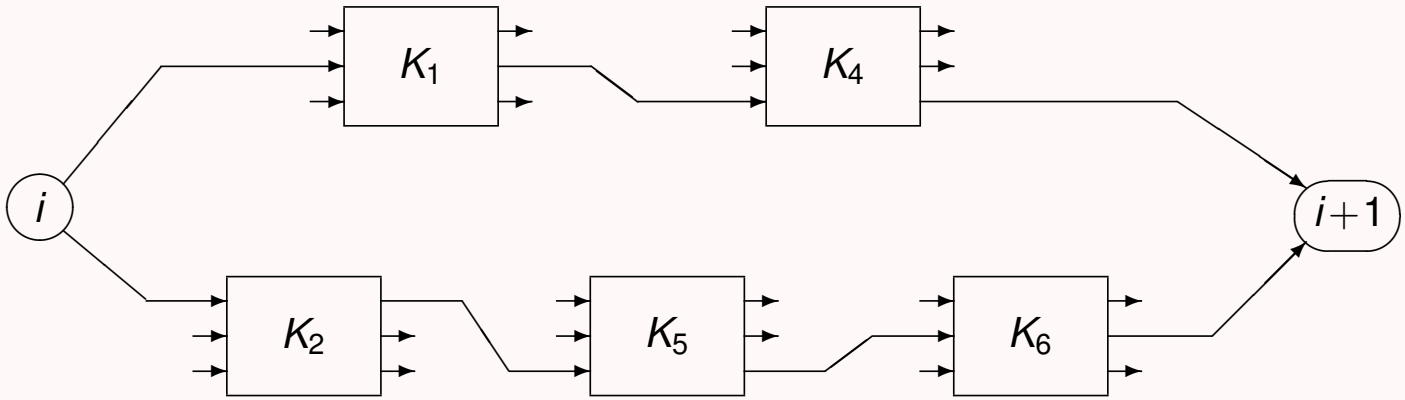
- Sei Graph K durch folgendes Symbol dargestellt:



- Der obere vom Knoten i ausgehende Weg orientiert sich an den Vorkommen von x_i in den Klauseln, der untere an denen von $\neg x_i$

Beweis:

- Konkretes Beispiel: Angenommen,
  x_i kommt in Klausel 1 an Position 2
  und in Klausel 4 an Position 3 vor
  $\neg x_i$ kommt in Klausel 2 an Position 1,
  in Klausel 5 an Position 3
  und in Klausel 6 an Position 2 vor.
  Dann sehen die beiden Verbindungen zwischen Knoten i und Knoten $i+1$ wie folgt aus:



Beweis:

- Da von allen Knoten $1, 2, \ldots, n$ jeweils derartige Pfade durch die Teilgraphen $K_1, \ldots, K_m$ ausgehen, sind schließlich alle 3 Eingänge und Ausgänge der K_i „angeschlossen“, und es gibt also keine „frei hängenden“ Kanten mehr.
- Nun noch zu beweisen:

$$F \text{ ist erfüllbar} \quad \Leftrightarrow \quad G \text{ hat Hamilton-Kreis}$$

Beweis:

- Wenn Formel F eine erfüllende Belegung besitzt, so kann beginnend bei Knoten 1 auf folgende Art einen Hamilton-Kreis durchlaufen werden:
  Wenn Variable x_i die Belegung 1 hat, so folge man von Knoten i aus dem oberen Pfad, sonst dem unteren.
  Danach durchläuft der Pfad die entsprechenden „Klauselgraphen“ K_j, in denen x_i (bzw. $\neg x_i$ vorkommt). Die Klauselgraphen müssen nun auf eine der drei o.a. Möglichkeiten durchlaufen werden
  (Wahl hängt davon ab, wie viele weitere Literale in der Klausel j den Wert 1 erhalten).
  Dies kann immer so arrangiert werden, dass bei Rückkehr zum Knoten 1 *alle* Knoten der Klauselgraphen K_j, $j = 1, \ldots, m$, durchlaufen wurden.
  Diese Methode liefert also einen Hamilton-Kreis.

Beweis:

- Sei umgekehrt angenommen, G besitze einen Hamilton-Kreis.
- Dieser Kreis durchläuft Knoten 1, dann gewisse K_j, Knoten 2, dann gewisse K_j, usw., bis er zum Knoten 1 zurückkehrt.
- Es ist dem Hamilton-Kreis nicht möglich, anders als vorgesehen durch die Graphen K_j zu passieren. (siehe *Zusatz-Satz*)
- Man definiere eine Variablen-Belegung für x_i anhand dessen, ob der Hamilton-Kreis den Knoten i nach oben (=1) oder nach unten (=0) verlässt.
- Belegung erfüllt F, denn jeder Klauselgraph wird mindestens einmal passiert – die entsprechende Klausel erhält also den Wert 1.

□

Zusatz-Satz

Wenn der K umgebende Graph G einen Hamilton-Kreis besitzt, so verläuft dieser, immer wenn er den Teilgraphen K passiert, folgendermaßen: Wenn der Hamilton-Pfad bei a (bzw. b bzw. c) in K hineinläuft, so verlässt er K bei A (bzw. B bzw. C).

Beweis:

- Angenommen, der Hamilton-Kreis betritt K beim Knoten a, verlässt K jedoch nicht beim Knoten A.
- Verschiedenen Möglichkeiten:
  Der Pfad verläuft danach durch die Knoten a, A, B und verlässt K bei B:
  Dann ist Knoten b in eine „Sackgasse“ geraten, man kann b zwar noch erreichen, den Hamilton-Kreis jedoch nicht mehr schließen.

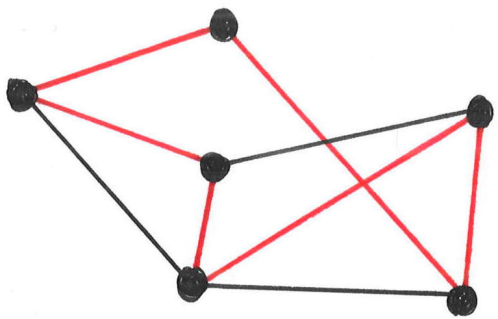
Beweis:

- *Fall* $a - A - B - C$: wieder Sackgasse bei b.
  Fall $a - c - C$: Knoten A ist nicht mehr erreichbar.
  Fall $a - c - b - B$: Knoten A und C nicht mehr erreichbar.
  Fall $a - c - b - B - C$: Knoten A nicht mehr erreichbar.
  Fall $a - c - C - A - B$: Sackgasse bei b.
  $\Rightarrow$ Alle Fälle erschöpft!
- Falls der Hamilton-Kreis bei b oder bei c nach K eintritt, so ist das Argument analog, da der Graph in a, b, c symmetrisch ist.
- Ein bei a eintretender Hamilton-Kreis kann also nur folgende Wege durch K nehmen kann: $a - A$, $a - c - C - A$ oder $a - c - b - B - C - A$. □

UNGERICHTETER HAMILTON-KREIS

gegeben: Ein ungerichteter Graph $G = (V, E)$.

gefragt: Besitzt G einen Hamilton-Kreis? (Dies ist eine Permutation π der Knotenindizes $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$, so dass für $i = 1, \ldots, n-1$ gilt: $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ und außerdem $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$).



UNGERICHTETER HAMILTON-KREIS

Satz

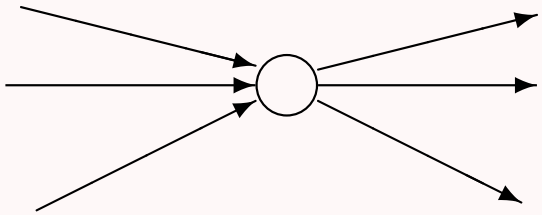
UNGERICHTETER HAMILTON-KREIS ist NP-vollständig.

Beweis:

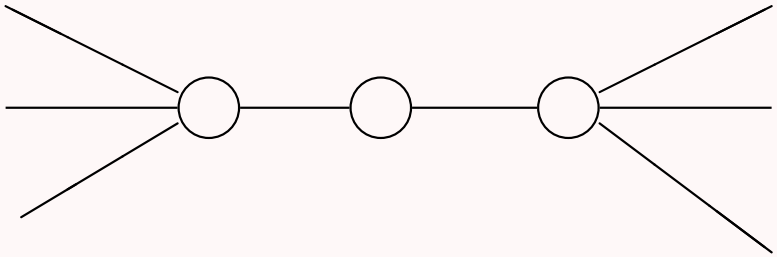
- Man zeigen, dass der gerichtete Fall auf den ungerichteten reduziert werden kann.
- Dazu ist anzugeben, wie man gerichtete Graphen in ungerichtete transformiert, so dass Hamilton-Kreis Eigenschaft erhalten bleibt.

Beweis:

- Jeder Knoten
  (mit gewissen hereinkommenden und herausgehenden Kanten)



wird lokal ersetzt durch drei Knoten:



Beweis:

- Wenn der gerichtete Graph einen Hamilton-Kreis besitzt, hat der ungerichtete auch einen entsprechenden.
- Wenn umgekehrt der ungerichtete Graph einen Hamilton-Kreis besitzt, dann ist es nicht möglich, dass der Kreis, nachdem er in den linken Knoten einer Dreier-Gruppe hereinkommt, diese wieder nach links verlässt.
  In diesem Fall würde der Kreis, wenn er den mittleren Knoten irgendwann erreicht, in einer „Sackgasse“ landen.
  Daher muss jeder Hamilton-Kreis, der von links in eine solche Dreier-Knotengruppe hineinläuft, diese nach rechts wieder verlassen.
- Deshalb lässt sich aus dem ungerichteten Hamilton-Kreis auch wieder ein entsprechender im gerichteten Graphen gewinnen. □

TRAVELLING SALESMAN

gegeben: Eine $n \times n$ Matrix $(M_{i,j})$ von „Entfernungen“ zwischen n „Städten“ und eine Zahl k.

gefragt: Gibt es eine Permutation π (eine „Rundreise“), so dass $\sum_{i=1}^{n-1} M_{\pi(i),\pi(i+1)} + M_{\pi(n),\pi(1)} \leq k$?

	A	B	C	D
A	–	20	10	33
B	21	–	18	10
C	10	18	–	5
D	33	10	-2	–

$n = 4$

$k = 40$

20
10
−2
10
38

Satz

TRAVELLING SALESMAN ist NP-vollständig.

Beweis:

- Man zeige *UNGERICHTETER HAMILTON-KREIS* $\leq_p$ *TRAVELLING SALESMAN*.
- Reduktion wird bewerkstelligt durch folgende Abbildung:

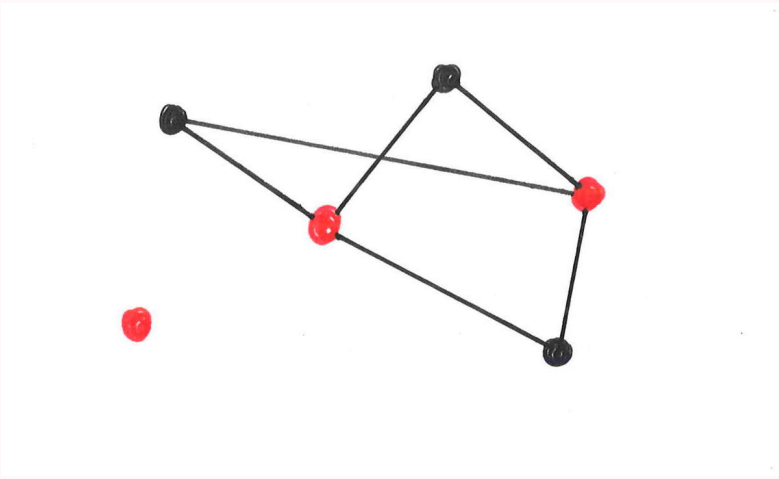
$$G = (\{1, \ldots, n\}, E) \mapsto \begin{cases} \text{Matrix:} & M_{ij} = \begin{cases} 1, & \{i,j\} \in E \\ 2, & \{i,j\} \notin E \end{cases} \\ \text{Rundreiselänge:} & n \end{cases}$$

□

FÄRBBARKEIT

gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in N$.

gefragt: Gibt es eine Färbung der Knoten in V mit k verschiedenen Farben, so dass keine zwei benachbarten Knoten in G dieselbe Farbe haben?



Satz

FÄRBBARKEIT ist NP-vollständig.

Beweis der NP-Härte

Zeige 3KNF $\leq_p$ FÄRBBARKEIT

Sei $F = K_1 \wedge \ldots \wedge K_m$ eine Formel in KNF mit *genau* drei Literalen pro Klausel und den Variablen $x_1, \ldots, x_n$.

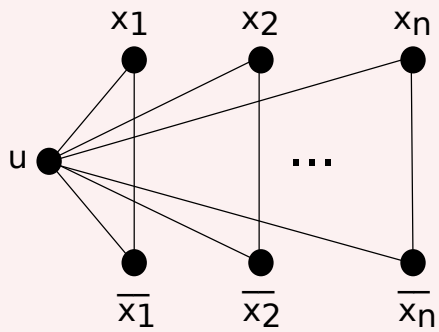
Konstruiere einen Graphen $G = (V, E)$ mit $|V| = 2n + 5m + 2$ der genau dann 3-färbbar ist, wenn F erfüllbar ist.

Teilgraph mit $2n + 1$ Knoten

Farben: $\{\text{ROT}, \text{WAHR}, \text{FALSCH}\}$.

Sei o.B.d.A. u ROT gefärbt, so kann x_i nur WAHR (FALSCH) gefärbt werden, wenn $\overline{x}_i$ FALSCH (WAHR) gefärbt wird.

Skizze



Beweis der NP-Härte

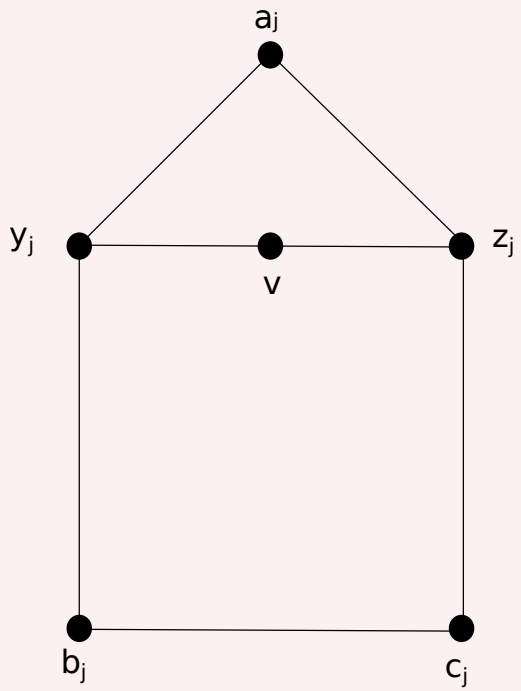
Klauselgraphen

Für jede Klausel K_j wird der rechts zu sehende Graph hinzugefügt, wobei der Knoten v für jeden Graphen K_j *derselbe* ist.

Kanten

Jeder Klausel-Graph hat ausgehende von a_j, b_j, c_j Kanten zu dem betreffenden Literal-Knoten im oberen Bild, entsprechend dem Vorkommen des Literals.

Skizze

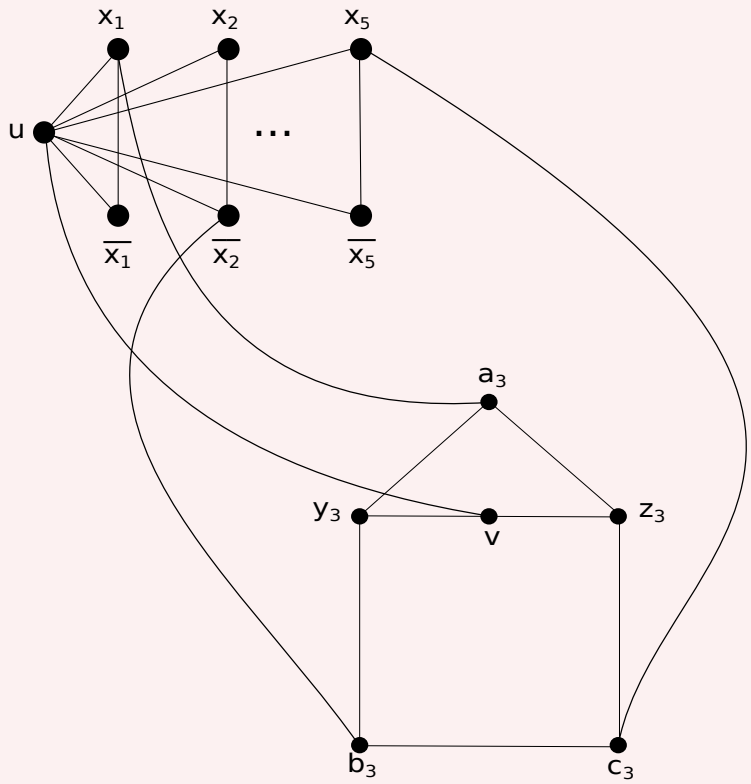


Beweis der NP-Härte

Beispiel

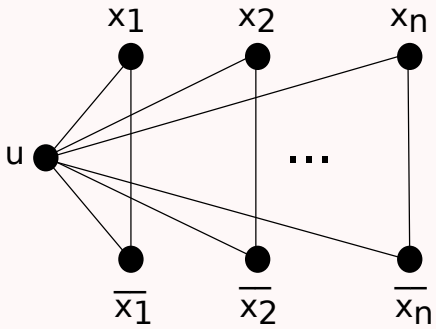
Klausel K_3 enthalte die Literale $x_1, \overline{x}_2, x_5$, so gibt es in G die Kanten $\{a_3, x_1\}, \{b_3, \overline{x}_2\}, \{c_3, x_5\}$, bildlich:

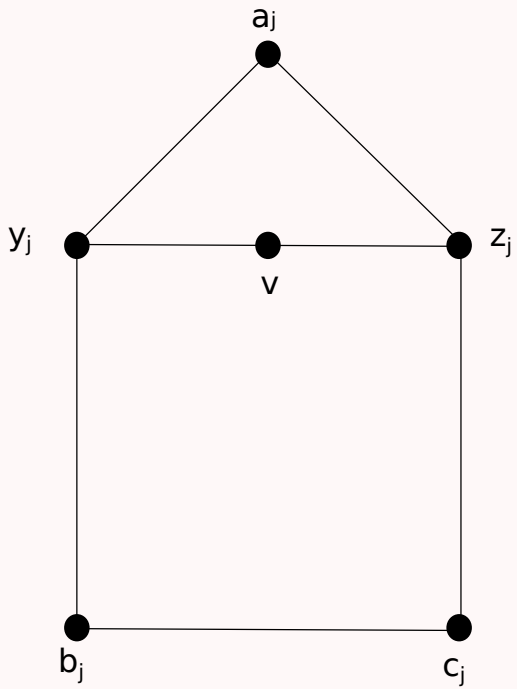
Skizze



Beweis der NP-Härte

Angenommen, F ist erfüllbar, dann kann u in G ROT, die Knoten $x_i, \overline{x}_i$ gemäß ihrer erfüllenden Belegung in WAHR oder Falsch eingefärbt werden.
Der Knoten v erhalte die Farbe FALSCH, dann kann jeder K_j-Graph zulässig eingefärbt werden, da nicht alle Nachbarknoten von a_j, b_j, c_j die Farbe FALSCH tragen (Fallunterscheidung).





Beweis der NP-Härte

Sei umgekehrt G mit 3 Farben färbbar und sei o.B.d.A u ROT. Nehmen wir an, eine Färbung entspricht keiner erfüllenden Belegung von F, so gibt es eine Klausel K_j, in der alle Literale die Farbe FALSCH haben. Deren Nachbarknoten in G, nämlich a_j, b_j, c_j im Klausel-Graphen K_j müssen dann ROT oder WAHR sein. b_j und c_j sind benachbart, daher kann nur der eine Knoten ROT und der andere WAHR sein. Daher können auch y_j und z_j nur WAHR und ROT sein. Das impliziert, dass a_j FALSCH sein muss, da a_j Nachbar von y_j und z_j ist, was einen Widerspruch darstellt. Die Färbung liefert uns also eine erfüllende Belegung von F.